**Reliable Security: Double Error Correction for AN Codes in Encryption Systems**

*Yuan-Fu Liu and Tsung-Chu Huang*

Department of Electronics Engineering, National Changhua University of Education, Changhua, Taiwan

*Abstract*—Reliable encryption is crucial for secure computing. Product (AN) Codes enable fault-tolerant encryption computations while conventional AN codes only correct single-bit errors. This paper proposes a double error correction (DEC) method for AN codes. Since [[1]](#ref1), no analytical decoding or systematic LUT-based approach for DEC of AN codes has been explored in the past fifty years. To address this, we introduce a trade-off algorithm combining software-based searching with a method that reduces area cost by a factor of *N*. Experimental results show that our method enhances encryption reliability while minimizing computational cost, making it suitable for cryptographic applications like RSA encryption.

# Introduction

In the era of artificial intelligence (AI), the rapid growth of data and computing needs has made big data processing an important topic. Encryption is key to protecting sensitive information, but any errors during the process can spread to later computations, risking the security of the data [[2]](#ref2). Solving these issues is crucial to building reliable and secure systems in today's AI-driven world. *大範圍的重要性*

RSA encryption [[3]](#ref3) is a common type of asymmetric encryption that requires heavy computation [[4]](#ref4)[[5]](#ref5). It processes data up to 2048 bits, making it essential for secure communication. The RSA encryption process relies on the selection of suitable prime numbers and involves intensive computations [[6]](#ref6), such as long integer multiplication and finding remainders. However, this process is vulnerable to fault injection attacks [[7]](#ref6), which can introduce critical errors, particularly arithmetic weight errors (AWE), compromising the overall security and reliability of the encryption system [[2]](#ref2). *可靠加密的重要性*

AN codes [[8]](#ref8), a type of arithmetic product code, offers error detection and correction capabilities while maintaining its functionality after arithmetic operations [[9]](#ref9). This distinguishes it from channel codes used in communication systems, such as BCH codes and RS codes [[10]](#ref10). However, conventional implementations are limited to detecting single-bit arithmetic weight errors, posing challenges to achieving higher reliability in encryption applications. *提到AN code, 並簡述1bit AN 可靠度不足*

In this paper, we propose a method to correct 2-bit AWE in AN codes, enhancing reliability compared to existing implementations. In Section II with a review of 2-bit error-correcting channel codes and the current methods for encoding and decoding AN codes. Section III details the proposed technique. In Section IV evaluates the improvements and compares the results. Finally, Section V concludes with a summary. *瀏覽II~V章*

# Previous work

## Channel codes 點出BCH RS codes的缺點

Previous works have shown that channel codes, such as BCH codes and RS codes [[10]](#ref10), are effective in detecting and correcting errors during data transmission. However, these codes are designed for channel-related errors and cannot address arithmetic errors that may arise during computational operations. Such limitations highlight the need for specialized error correction methods, like AN codes, to handle arithmetic errors effectively.

## Arithmetic Weight Error Model 算術錯誤模型

An *n*-bit 2’s complemented binary-coded binary (BCB) number *x* can represented as

, (1)

where. It can be also represented by a TCB number,

, (2)

where and signs ± separately represent ±1. Here will be called a *digital weight* (DW) for distinguishing with *weights* inneural network (NN) and *arithmetic weight* (AW) of *x*. AW is defined as the minimum count of ± signs for representing, namely,

. (3)

The AW distance between two integers *x* and *y*, ​is defined as , where ***w*** is the arithmetic weight. A number with AW ≤ 2 (3) is called nice (light) number respectively. The AW-minimization algorithm [[11]](#ref111) reduces the AW distance efficiently.

## AN Codes 介紹AN Codes

The AN codes take the residue for error location. In a () AN codes, the message word *N*, , is encoded to the product codeword *C*=*AN*, possibly infected by an AWE, as , and decoded by . where . As illustrated in Fig.1, the *w*-AWE, *e*, can be located as , where denotes the residue of *x* mod *A*. Note that in the AWE model a single error can be or at bit location *i* therefore a subgroup and a coset are generated by × 2 from +1 and −1 ≡ A − 1 for generating distinct residues for identifying the error location. For example, a checkbits locates an error at index 3 of a Hamming codeword, while a residue (mod 29) denotes an error at location 3 of the 29N codes.



Fig. 1 Comparison of Hamming/AN Codes in error location.

If *A* is an odd prime number, and *2* is a primitive element for generating a multiplicative group G(*A*) of Galois field, GF(*A*), then G(*A*) is perfect to locate the AWE indices of a -bit *A*N codeword. Since it is easy for today’s computer, for applications only, we will search for a good multiplier number A for AN codes by an exhaustive searching scheme without too many complicated theories. Table I shows the searched AN-code multipliers A Row ***w***(A) shows the AW of A. For example, since GF(29)={0; (0, 1, 2, 4, 8, 16, 3, 6, 12, 24, 19, 9, 18, 7, 14, 28, 27, 25, 21, 13, 26, 23, 17, 5, 10, 20, 11, 22, 15)}, 29 is a good multiplier for a perfect AN codes with single error correction (SEC) capability of a 14-bit codeword for .

1. Multipliers for perfect AN codes

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *n* | 5 | 6 | 9 | 11 | 14 | 18 | 23 | 26 | 29 | 30 | 33 | 35 | 39 | 41 | 50 | 51 | 53 | 55 | 65 |
| A | 11 | 13 | 19 | 23 | 29 | 37 | 47 | 53 | 59 | 61 | 67 | 71 | 79 | 83 | 101 | 103 | 107 | 121 | 131 |
| ***w***(A) | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 3 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 3 | 3 |

To analyze the differences and applications of channel codes and AN codes, we propose an transmission fault module, as illustrated in Figure 2. After passing through the channel encoder, data undergoes 10 transmission stages, with errors potentially occurring at any stage. The error correction capabilities of channel codes and AN codes across these stages have been evaluated and compared in Table II.



Fig. 2 Proposed transmission fault module.

*提出錯誤傳輸模型*

As shown in Table II, AN codes can serve as an alternative to channel codes for error correction (i.e., ②). Additionally, integrating AN codes into the channel encoder and decoder enhances fault tolerance by preventing errors during the encoding and decoding processes (i.e., ①and ③).

1. an codes v.s. channel codes

|  |  |  |
| --- | --- | --- |
|  | Not enable to detect and correct errors. | Enable to detect and correct errors. |
| Channel Codes | ①③④⑤⑥⑦⑧⑨⑩ | ② |
| AN Codes | ④⑤⑨⑩ | ①②③⑥⑦⑧ |

Due to the limitations of Galois fields, the described AN code has a drawback: it can only detect and correct single-bit arithmetic weight errors and lacks the capability for double error correction (DEC), making it unable to handle two-bit or higher arithmetic weight errors. When the most significant bit (MSB) is prone to errors, the second MSB is often similarly vulnerable. To improve reliability, the implementation of DEC is essential. *提出現行AN Code沒辦法做DEC*

# Proposed technique

## Look-up table(LUT) based on DEC

To evaluate the suitability of a number *A* as an AN codes capable of implementing DEC, Figure 3 illustrates the proposed method for identifying a suitable AN codes to implement DEC. First, the data of specific bits (*n*-bits) is multiplied by *A*. The data multiplied by *A* is referred to as *AN* (*N*-bits). Next, single and double AWE are generated and added to the *AN* to simulate error data. The error data is then divided by *A*, and the remainders are analyzed. If all remainders are distinct for the error data, *A* is deemed suitable for detecting and correcting single and double AWE in the specified bits.

*描述搜尋*DEC A*的過程*



Fig. 3 The flowchart for searching n-bits DEC AN codes.

Figure 4 presents the data range corresponding to different bit widths along with the identified A values. The results indicate that as the bit width increases, the proportion of A within the data decreases, making its overhead negligible for large integers. This observation supports the feasibility of applying DEC to encryption technology that involve large-scale computations, such as RSA. *列出10~18bits DEC A*

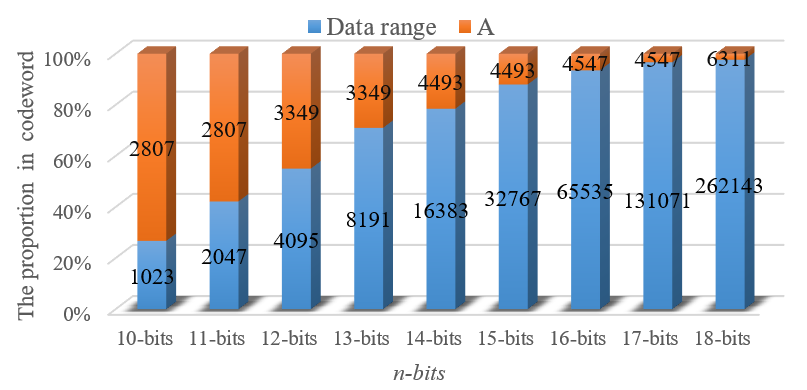


Fig. 4 The proportion of A in codeword.

*描繪LUT的圖*

The identified *A* ensures distinct modulus values for any error data. Leveraging this property, a one-to-one error *LUT* can be constructed to correct single and double AWE, as demonstrated in Figure 5. As illustrated in the figure, the *LUT* for DEC comprises two components: the single AWE *LUT* and the double AWE *LUT*. The size of the double AWE *LUT* is approximately *N* times that of the single AWE *LUT*, where *N* represents the bit width of the error data (i.e., the bit width of *AN*), contributing significantly to area overhead. To address this issue, the following section presents an alternative approach that achieves DEC using only the single AWE LUT, effectively reducing area costs.



Fig. 5 Look-up table (LUT) for DEC.

## A Trade-off Method For DEC

A *LUT-based* approach (called *Parallel*) for DEC provides fast and efficient error correction but incurs excessive area overhead, making it unsuitable for VLSI implementation. Alternatively, if error correction is performed without utilizing a *LUT* and instead relies on a purely software-based approach (called *Sequential*), the process would involve multiple iterations to compute, compare, and correct all possible AWE values. While this method minimizes area overhead, it significantly increases the search and computation time, making it inefficient for practical implementation. To achieve a balance between efficiency and resource utilization, we propose a *Trade-off* approach that combines software-based searching with a single AWE *LUT* for error correction, as illustrated in Figure 6



Fig. 6 *Trade-off* algorithm.

In the proposed algorithm, **SEC\_LUT** represents the single AWE *LUT*. The trade-off approach leverages only the single AWE *LUT*, achieving an area reduction by a factor of *N*, thereby enhancing feasibility for VLSI implementation. Moreover, this method eliminates the need for extensive loop iterations and per-AWE computations, effectively reducing computational time overhead.

## Automatic Synthesizer

Building on the *Trade-off* method, a Python-based synthesizer was developed to generate *LUT* for different AWE. This synthesizer converts the *LUT* from a *.py (python)* file into a *.v (verilog)* file, as shown in Figure 7



Fig. 7 Synthesizer flow of LUT.

After implementing the required *LUT* functionality in Python, the synthesizer automatically generates the corresponding Hardware Description Language (HDL) code, facilitating seamless hardware synthesis. This enables the rapid development of fault-tolerant hardware designs with DEC capabilities, making them readily deployable on FPGA or ASIC platforms in the future.

# Evaluation and Experimental Results

To evaluate the cost of the three methods-*Parallel, Sequential*, and *Trade-off*-we assume a fixed total cost for both time and area. The area cost is defined based on the bit width of the data, where *n* represents the input data bit width, and *N* denotes the bit width of the data after multiplication by *A* (i.e., *AN*). For time cost estimation, a single iteration of a for-loop is assigned a unit cost of 1, while arithmetic operations and comparisons incur a time cost of *N*. The *LUT* decoding time for AWE is approximated as *log2*(*number of input line*). Based on these cost assumptions, Table III summarizes the comparative analysis.

1. The cost table of each method

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | **Parallel** | | **Sequential** | | **Trade-off** | |
| ***n*** | ***N*** | ***A*** | ***Area:***  *2****(N)3*** | ***Time:***  ***log2(2N2)*** | ***Area:***  *C* | ***Time:***  *2****(N)2N*** | ***Area:***  *2****(N)2*** | ***Time:***  ***2Nlog2(2N)*** |
| 4 | 14 | 665 | 5488 | 9 | *C=9* | 5488 | 392 | 135 |
| 8 | 19 | 1939 | 13718 | 10 | *C=10* | 13718 | 722 | 200 |
| 12 | 24 | 3349 | 27648 | 11 | *C=11* | 27648 | 1152 | 268 |
| 16 | 29 | 4547 | 48778 | 11 | *C=11* | 48778 | 1682 | 340 |
| 20 | 33 | 6311 | 71874 | 12 | *C=12* | 71874 | 2178 | 399 |
| 24 | 38 | 13837 | 109744 | 12 | *C=12* | 109744 | 2888 | 475 |
| 28 | 43 | 17619 | 159014 | 12 | *C=12* | 159014 | 3698 | 553 |
| 30 | 45 | 18613 | 182250 | 12 | *C=12* | 182250 | 4050 | 584 |

In the **Sequential area cost** column of Table III, the software execution time cost is initially unknown and is therefore defined as a constant *C*. However, based on the principle of maintaining a fixed total cost, the **Sequential area cost** can be approximated as equivalent to the **Parallel time cost**. As shown in the table, the ***Trade-off*** approach achieves a more balanced distribution of time and area costs, preventing extreme cost values that would otherwise pose implementation challenges.

To highlight the cost differences among the methods more clearly, the data from Table III is visualized as a curve plot in Figure 8, where the horizontal axis represents the bit width and the vertical axis denotes the corresponding cost.

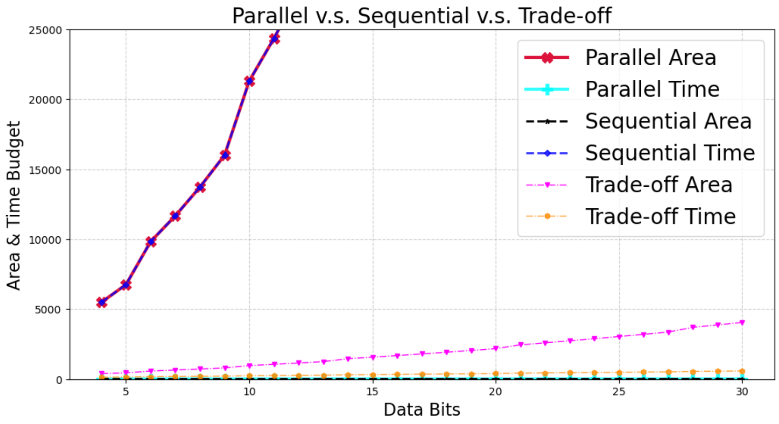


Fig. 8 The cost curve of each method

As shown in the results, the **Parallel area cost** and **Sequential time cost** (red and blue lines) exhibit an approximately exponential growth as the bit width increases, exceeding the graph's range beyond 10 bits. In contrast, the ***Trade-off*** approach (orange and magenta lines) demonstrates a gradual linear increase in both area and time costs. This balanced approach optimizes computational efficiency and hardware resource utilization, making it highly suitable for cryptographic applications involving large integer computations, such as RSA. It achieves fault tolerance while maintaining cost-effectiveness.

##### References

1. C.K. Liu. "CKLiu1972\_Error Correction Codes in Computer Arithmetic," technical report , University of Illinois at Urbana-Champaign, 1972.
2. D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in \*Advances in Cryptology - EUROCRYPT '97\*, Lecture Notes in Computer Science, vol. 1233, W. Fumy, Ed. Berlin, Germany: Springer-Verlag, 1997, pp. 37–51.
3. W. Stallings, "Cryptography and Network Security: Principles and Practice," 7th ed., Pearson, 2016.
4. S. Y. Sun Yile and W. X. Wu Xingjun, "An area efficient modular arithmetic processor," 2003 5th International Conference on ASIC Proceedings, Beijing, China, 2003, pp. 1273-1276 Vol.2
5. Xin Zhou and Xiaofei Tang, "Research and implementation of RSA algorithm for encryption and decryption," Proceedings of 2011 6th International Forum on Strategic Technology, Harbin, Heilongjiang, 2011, pp. 1118-1121
6. R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM, vol. 21, no. 2, pp. 120–126, 1978.
7. M. Joye and M. Tunstall,"Fault Analysis in Cryptography," Springer, 2012.
8. J. M. Diamond, "Checking codes for digital computers," Proc. IRE, vol. 43, pp. 487-488, Apr. 1955.
9. D. T. Brown, "Error detecting and error correcting binary codes for arithmetic operations," IRE Trans. Electron. Comput., vol. EC-9, pp. 333-337, Mar. 1960
10. R. W. Hamming, "Error Detecting and Error Correcting Codes," Bell System Technical Journal, vol. 29, no. 2, pp. 147–160, 1950.
11. C.-D. Tsai, T.-Y. Chen, H.-W. Fu and T.-C. Huang. "TCBNN: Error-Correctable Ternary-Coded Binarized Neural Network," 2021 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS2021), virtual, June 7, 2021.