A Novel Mathematical Formal Proof in Zhang-Wang's Cryptographic Algorithm

Junwei Yang[1], Xiangkun Tong[1], Chenglian Liu[1,*], Sonia C-I Chen[2]

[1]*School of Computing, Neusoft Institute of Guangdong, Foshan, Guangdong, 528225, China.*

[2]*School of Economics, Qingdao University, Qingdao, Shandong, 266061, China.*

**Corresponding Author.*

Abstract

Formal verification is to use mathematical methods to prove that our scheme is correct. This scheme is just a pronoun. It may be expressed as a hardware, a software or an algorithm. Errors in hardware are more difficult to modify than errors in software, so formal proofs and inspections often appear in the argument for hardware design. But, it does not mean that the software does not need to be formally verified. In addition to digital circuits or combinational circuits, cryptographic protocols also need to be formally verified. Formal proof can only ensure whether the result of logical inference is consistent with the previous stage, and can not guarantee whether there are defects in the process of logical inference. In this article the authors take as an example of Zhang-Wang's digital signature algorithm, and point out two formal proof methods of Boolean algebra and Galois field respectively.

Keywords*: Exclusive-OR, Formal Verification, Logical Defect,*

I. Introduction

An El Gamal-like signature scheme was proposed by Zhang and Wang in 2005, they did not used one-way hash functions and message redundancy. Subsequent relevant studies, such as forgery attacks by Yang and Hsu [2], and Guo-Li-Hu [3] in 2008. The Boolean or logic operation issue by Liu and Zhang [4], and Zheng [5]. Luo et al. [6] simply revealed algebra attack [7], [8] on another point of view. The operation of precedence topic by Xu et al. [9]. Yang and Liu [10], Qiu et al. [11] focus on probabilities; and others discussion i.e reverse engineering topic be talked by Luo et al. [12]. In this paper the authors will enumerate other viewpoint which demonstrate a formal proof [13] to the Zhang-Wang's algorithm. The section 2 review of Zhang-Wang algorithm, we use Java language to implementation it and check the experiment data. The section 3 we give the lemmas and theorem to prove our viewpoint, we also use Python language to examine our assumption. Fortunately, for beginners, our readers do not have to install Java or Python environments. Anyone can copy and paste our source codes by online compilers such as "https://www.jdoodle.com" to check the result. The relevant literatures shown on Table 1.

Table 1 Related Literatures

Year	Forgery Attack	Boolean	Precedence	Logic	Probability	Others
2008	Yang & Hsu [2]					
2008	Guo et al. [3]					
2009		Liu & Zhang [4]				
2010			Xu et al. [9]			
2012		Liu et al. [14]				
2014				Zheng [5]		
2014					Yang & Liu [10]	
2018					Qiu et al. [11]	
2019	Lou et al. [6]					
2021						Lou et al. [6]

II. Review of Zhang-Wang Algorithm

Notation:
p: denote the prime where it may 1024 bits length.
g: denote a primitive root.
y: denote the public key.
x: denote the private key.
M: denote the digitize message.
k: denote the nonce value.
$\oplus$: denote the exclusive-OR (XOR) operation.
$()_2$: denote binary number.
$()_{10}$: denote decimal number.

In 2005, Zhang and Wang [1] presented a signature algorithm without using one-way hash functions. The concepts are two parameters p and g, while p is a large prime where $g \in GF(p)$ is a primitive root of modulo p in this scheme. Alice selects her private key x, where $\gcd(x, p-1) = 1$, and calculates the public key

$$y \equiv g^x \pmod{p}. \tag{1}$$

This scheme contains of two steps: key generation and verification phases. The details are as follows.

2.1 Zhang-Wang Algorithm

1.Signature Generation Phase:
Supposing Alice signs the M, and she then runs as follow steps:
Step 1. Alice calculates

$$s \equiv (y + M)^{M \bmod p-1} \pmod{p}. \tag{2}$$

Step 2. Alice chooses a random integer

$$k \in Z_{p-1}^*, \tag{3}$$

and computes

$$r \equiv M \cdot s \cdot g^{-k} \pmod{p}. \tag{4}$$

Step 3. Alice calculates t since

$$s + t \equiv x^{-1} \cdot [(k - (r \oplus s)] \pmod{p-1}. \tag{5}$$

Step 4. Alice transmits the triple parameters (s, r, t) of M to the verifier.

2. Verification Phase:
After getting the parameters (s, r, t) transmitted by Alice, the verifier Bob checks follow steps to validate signature.
Step 1. Bob first calculates

$$\begin{aligned} M' &\equiv y^{s+t} \cdot r \cdot g^{r \oplus s} \cdot s^{-1} \pmod{p} \\ &\equiv g^{k-r \oplus s} \cdot M \cdot g^{-k+r \oplus s} \pmod{p} \\ &\equiv M \pmod{p}. \end{aligned} \tag{6}$$

Step 2. Bob verifies $s \equiv (y + M)^{M \bmod p-1} \pmod{p}$, if Equation (2) holds and the authenticity of the initial signature is verified.

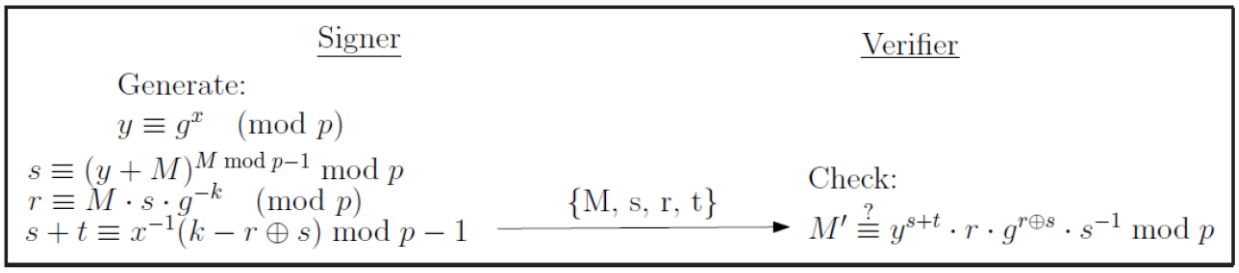

Figure 1. The Zhang-Wang Algorithm Scheme.

The concept and operating principle showed in Figure 1. It has simple express each parameter and role in the protocol clearly.

2.2 Coding by Language Java

Listing1 **ZhangWang.java**

```
import java.math.BigInteger;
import java.util.Scanner;

public class ZhangWangSignatureScheme {

  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.println("Chooses two public parameters p and g");
    BigInteger one = new BigInteger("1");
    BigInteger p = new BigInteger(scanner.next());
    BigInteger g = new BigInteger(scanner.next());
    // check p is prime
    if (p.isProbablePrime(1)) {
      System.out.println("p = " + p);
      System.out.println("g = " + g);
      System.out.println("The signer chooses a private key x");
      BigInteger x = new BigInteger(scanner.next());
      // check gcd(x, p − 1) = 1
      if (x.gcd(p.subtract(one)).equals(one)) {
        System.out.println("x = " + x);
        BigInteger y = g.modPow(x, p);
        System.out.println("y = " + y);
        System.out.println("Enter the message M that you wants to sign\nPlease enter a number to simplify the
calculation");
        BigInteger M = new BigInteger(scanner.next());
        System.out.println("M = " + M);
        BigInteger s = (y.add(M)).modPow(M.mod(p.subtract(one)), p);
        System.out.println("s = " + s);
        System.out.println("Chooses a number k");
        BigInteger k = new BigInteger(scanner.next());
        System.out.println("k = " + k);
        BigInteger gk = g.modPow(k, p);
        BigInteger InverseElementgk = inverse(gk, p);
        System.out.println("g^k * g^(-k) = 1 (mod p)\t" + gk + " * " + InverseElementgk + " = 1 (mod " + p + ")" );
        BigInteger r = ((M.multiply(s)).mod(p)).multiply(InverseElementgk)).mod(p);
        System.out.println("r = " + r);
        BigInteger InverseElementX = inverse(x, p.subtract(one));
        System.out.println("x * x^(-1) = 1 (mod p-1)\t" + x + " * " + InverseElementX + " = 1 (mod " + p + "-1)");
        BigInteger sADDt = (InverseElementX.multiply(k.subtract(r.xor(s)))).mod(p.subtract(one));
        System.out.println("xor(r, s) = " + r.xor(s));
        System.out.println("s + t = " + sADDt);
        BigInteger InverseElementS = inverse(s, p);
        System.out.println("s * s^(-1) = 1 (mod p)\t\t" + s + " * " + InverseElementS + " = 1 (mod " + p + ")");
        BigInteger MM = ((y.modPow(sADDt, p)).multiply(r).multiply(g.modPow(r.xor(s),
p)).multiply(InverseElementS)).mod(p);
        System.out.println("M' = " + MM);
      }else {
        System.out.println("gcd(x, p − 1) != 1");
      }
    }else {
      System.out.println("p is not prime number");
    }

    scanner.close();
  }

  // inverse
  public static BigInteger inverse(BigInteger x, BigInteger p) {
    BigInteger temp = new BigInteger("1");
    BigInteger one = new BigInteger("1");
    while (true) {
      if (temp.multiply(x).mod(p).equals(one)) {
        return temp;
      }
      temp = temp.add(one);
    }
  }

}
```

The authors follow that conception and algorithm to implementation the scheme by language Java.

2.3 The Experiment and Testing

We randomly assume the parameters g = 61, x = 49, p =191, to find the public key y as

$$140 \equiv 61^{49} \pmod{191}. \tag{7}$$

Given m = 38, we obtained

$$s \equiv 184 \equiv (140 + 38)^{38} \pmod{191}. \tag{8}$$

Let k = 51, where

$$r \equiv 6 \equiv 38 \cdot 184 \cdot 61^{-51} \pmod{191}. \tag{9}$$

By Equation (5), as known

$$184 + t \equiv 49^{-1} \cdot [(51 - (6 \oplus 184)] \pmod{190}, \tag{10}$$

then get t = 135. Finally, we calculate

$$\begin{aligned} 38 &\stackrel{?}{\equiv} 140^{184+135} \cdot 6 \cdot 61^{6 \oplus 184} \cdot 184^{-1} \pmod{191} \\ &\equiv 140^{319} \cdot 6 \cdot 61^{190} \cdot 109 \pmod{191} \\ &\equiv 83 \cdot 6 \cdot 1 \cdot 109 \pmod{191} \\ &\equiv 38 \pmod{191}. \end{aligned} \tag{11}$$

From Equation (7) to (11), the verifier can recovery and check the message by signatures (s, r, t), please see the experiment result in Figure 2.

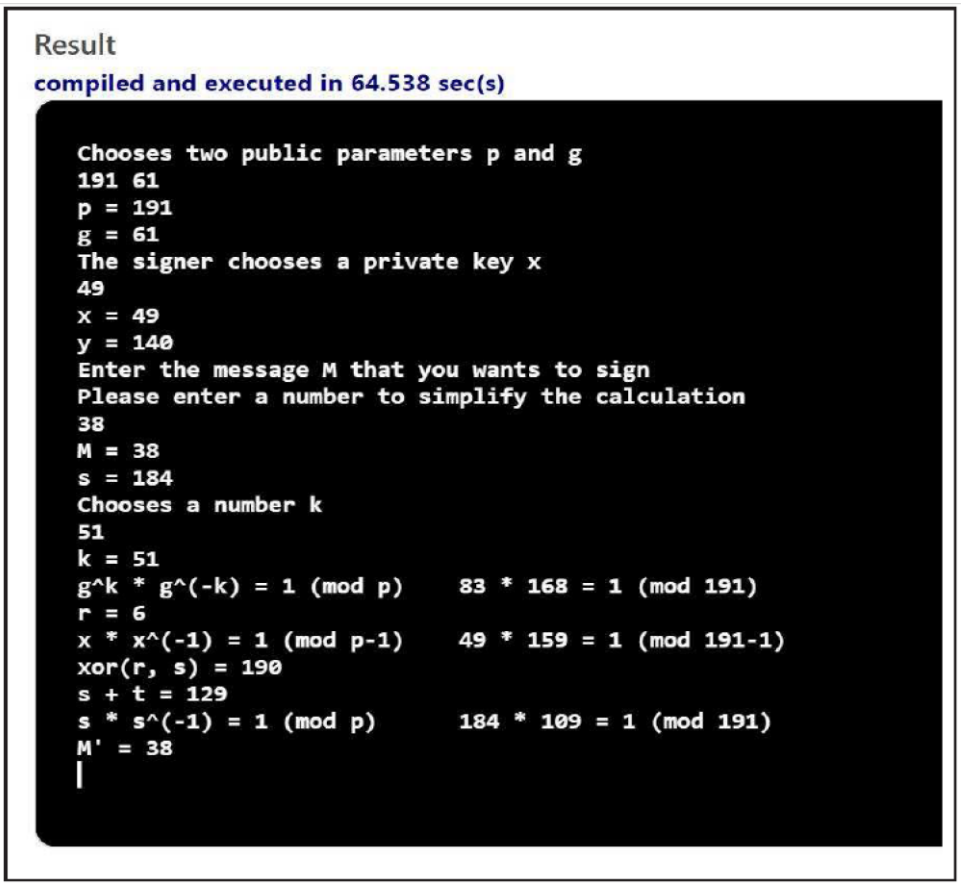

Figure 2. The result of the program executing by ZhangWang.java.

III. Security Analysis of Formal Proof

In this section, the authors would like to describe some points such as the logic defect, i.e the XOR extension problem [15]. Some people in related research also gave good contributions [16], [17], but different what we discussed here.

Exclusive-OR or exclusive disjunction is a logical operation. When the input has an odd number of 1s, the output is 1; when the input has an even number of 1s, the output is 0, In Figure 3, the authors express T is "1", namely true, and the F means to "0" as false, see Figure 3.

input		output
A	B	
F	F	F
F	T	T
T	F	T
T	T	F

Figure 3. The XOR Truth Table.

3.1 The Boolean Algebra Expression

The XOR operation can be expressed as

$$A \oplus B = (\neg A \wedge B) \vee (A \wedge \neg B), \tag{12}$$

or express to

$$B \oplus A = B\overline{A} + \overline{B}A. \tag{13}$$

In text book of digital circuit or logic circuit, it is the beginning knowledge. The XOR usually applied for adder, cryptosystem, image process and other applications. We can infer the Lemma 1 to Lemma 4 [4, 14].

Lemma 1. Assume $\oplus$ be a binary operation the set X. If $A \oplus B = B \oplus A$ for all $A, B \in X$. We said, it satisfies commutative law.

Proof: To prove $A \oplus B = \overline{A}B + A\overline{B}$ where $B \oplus A = B\overline{A} + \overline{B}A$, therefore $\overline{A}B + A\overline{B} = B\overline{A} + \overline{B}A$. We obtain $A \oplus B = B \oplus A$. Here, the XOR satisfies commutative rule.

Lemma 2. Assume $\oplus$ be a binary operation the set X. If $(A \oplus B) \oplus C = A \oplus (B \oplus C)$ for all $A, B \in X$. We said, it satisfies associative rule.

Proof: $(A \oplus B) \oplus C = (\overline{A}B + A\overline{B}) \oplus C.$
$(\overline{A}B + A\overline{B}) \oplus C.$
$= \overline{(A\overline{B} + \overline{A}B)}C + (A\overline{B} + \overline{A}B)\overline{C}.$
$= \overline{(A\overline{B} + \overline{A}B)}C + (A\overline{B} + \overline{A}B)\overline{C}.$
$= \overline{(\overline{A}B) \cdot (A\overline{B})}C + \overline{A}B\overline{C} + A\overline{B} \cdot \overline{C}.$
$= (A + \overline{B})(\overline{A} + B)C + \overline{A}B\overline{C} + A\overline{B} \cdot \overline{C}.$
$= A\overline{A}C + ABC + \overline{B} \cdot \overline{A}C + \overline{B}BC + \overline{A}B\overline{C} + A\overline{B} \cdot \overline{C}.$
$A\overline{A} = 0$ and $B\overline{B} = 0.$
$= ABC + \overline{B} \cdot \overline{A}C + \overline{A}B\overline{C} + A\overline{B} \cdot \overline{C}.$
Computing $A \oplus (B \oplus C)$
$= A \oplus (B \oplus C) = A \oplus (\overline{B}C + B\overline{C}).$
$= \overline{A}(\overline{B}C + B\overline{C}) + A\overline{(\overline{B}C + B\overline{C})}.$
$= ABC + ABC + A\overline{(\overline{B}C)} \cdot \overline{(B\overline{C})}.$

$= \overline{A} \cdot \overline{B}C + \overline{A}B\overline{C} + A(B + \overline{C})(\overline{B} + C).$
$= \overline{A} \cdot \overline{B}C + \overline{A}B\overline{C} + AB\overline{B} + A\overline{B} \cdot \overline{C} + ACB + A\overline{C}C.$
$= \overline{A} \cdot \overline{B}C + \overline{A}B\overline{C} + ABC + A\overline{C} \cdot \overline{B}.$
$\because ABC + \overline{B} \cdot \overline{A}C + \overline{A}B\overline{C} + A\overline{B} \cdot \overline{C} = \overline{A} \cdot \overline{B}C + \overline{A}B\overline{C} + ABC + A\overline{C} \cdot \overline{B}.$
$\therefore (A \oplus B) \oplus C = A \oplus (B \oplus C).$
Here, the XOR also matches associative rule.

Lemma 3. Let $A = B$, $A \oplus B = \overbrace{0000 \ldots 0000}^{bits}$.

Proof: As known from Figure 3, we get $A \oplus A = 0$, therefore $A \oplus B = \overbrace{0000 \ldots 0000}^{bits}$.

Lemma 4. If A and B are both odd numbers, where $(A) \oplus (-A) = \overbrace{1111 \ldots 1110}^{bits}$, $(B) \oplus (-B) = \overbrace{1111 \ldots 1110}^{bits}$, then $(A \oplus B) = (-A \oplus -B)$.

Proof: According to Lemma 3, if $A = B$, then $(A \oplus B) \oplus (-A \oplus -B) = \overbrace{0000 \ldots 0000}^{bits}$.
From Lemma 1 and Lemma 2, we rewrite this equation $(A \oplus B) \oplus (-A \oplus -B) = (A \oplus -A) \oplus (B \oplus -B)$.

According to Lemma 4, $A \oplus -A = B \oplus -B$. From Lemma 3, $(A \oplus B) \oplus (-A \oplus -B) = \overbrace{0000 \ldots 0000}^{bits}$.\\
$\therefore (A \oplus B) = (-A \oplus -B)$.
For Example:

$$\begin{aligned}
(66)_{10} &= (01000010)_2 \\
(62)_{10} &= (00111110)_2 \\
(66)_{10} \oplus (62)_{10} &= (124)_{10} = (01111100)_2 \\
(-66)_{10} &= (1111111110111110)_2 \\
(-62)_{10} &= (1111111111000010)_2 \\
(-66)_{10} \oplus (-62)_{10} &= (124)_{10} = (01111100)_2
\end{aligned} \tag{14}$$

The pretender Eve can falsify the valid parameters (r, s) where

$$(\mathrm{r} \oplus s) = (-r \oplus -s). \tag{15}$$

She does three steps as follow:
Step 1. Eve lets

$$r' \equiv -r. \tag{16}$$

Step 2. Eve lets

$$s' \equiv -s. \tag{17}$$

Step 3. Eve computes

$$s + t \equiv s' + t'. \tag{18}$$

Proof:

$$\begin{aligned}
M' &\equiv y^{s'+t'} \cdot r \cdot g^{r' \oplus s'} \cdot (s')^{-1} \pmod{p} \\
&\equiv g^{k - r' \oplus s'} \cdot M \cdot g^{-k} \cdot g^{r' \oplus s'} \pmod{p} \\
&\equiv M \pmod{p}.
\end{aligned} \tag{19}$$

By Lemma 1 to Lemma 4, we obtained "$r \oplus s = r' \oplus s'$" if r and s both are odd or even integers where they satisfied specified rules. Eve may forge successfully for her action.

3.2 The Galois Field Expression

From above, the proof of Lemma 1 to Lemma 4 is described by Boolean algebra. In this paragraph, the authors borrowed binary concept which it re-express to Galios field form, and one theorem integrated four lemmas.

Theorem 1. If $2^m \| A$, $2^m \| B$, then $(A \oplus B) \equiv (-A \oplus -B) \bmod 2^n$, since $m < n \in N$, $A \in N$ and $B \in N$.

Proof: As know $2^m \| A$, $2^m \| B$, we get $A = a_i \sum_{i=m+1}^{n-1} 2^i + 2^m$, $B = b_i \sum_{i=m+1}^{n-1} 2^i + 2^m$ where $a_i \in GF(2)$, $b_i \in GF(2)$.

Suppose $-A \equiv (2^2 - A) \bmod 2^n$, namely

$$\begin{aligned} & -A \equiv (2^n - A) \bmod 2^n \\ & \equiv (\sum_{i=0}^{n-1} 2^i + 1 - A) \bmod 2^n \\ & \equiv (\sum_{i=0}^{n-1} 2^i - a_i \sum_{i=m+1}^{n-1} 2^i - 2^m + 1) \bmod 2^n \\ & \equiv (\bar{a}_i \sum_{i=m+1}^{n-1} 2^i + \sum_{i=0}^{m} 2^i - 2^m + 1) \bmod 2^n \\ & \equiv (\bar{a}_i \sum_{i=m+1}^{n-1} 2^i + 2^m) \bmod 2^n. \end{aligned} \tag{20}$$

Similarly, $-B \equiv (\bar{b}_i \sum_{i=m+1}^{n-1} 2^i + 2^m) \bmod 2^n$. We express $(A \oplus B) \equiv \left((a_i \oplus b_i) \sum_{i=m+1}^{n-1} 2^i \right) \bmod 2^n$, and rewrite as

$$\begin{aligned} & (-A \oplus -B) \\ & \equiv \left((\bar{a}_i \sum_{i=m+1}^{n-1} 2^i + 2^m) \oplus (\bar{b}_i \sum_{i=m+1}^{n-1} 2^i + 2^m) \right) \bmod 2^n \\ & \equiv \left((\bar{a}_i \sum_{i=m+1}^{n-1} 2^i) \oplus (\bar{b}_i \sum_{i=m+1}^{n-1} 2^i) + (2^m \oplus 2^m) \right) \bmod 2^n \\ & \equiv \left((\bar{a}_i \oplus \bar{b}_i) \sum_{i=m+1}^{n-1} 2^i \right) \bmod 2^n \\ & \equiv \left((a_i \oplus b_i) \sum_{i=m+1}^{n-1} 2^i \right) \bmod 2^n \\ & \equiv (A \oplus B) \bmod 2^n. \end{aligned} \tag{21}$$

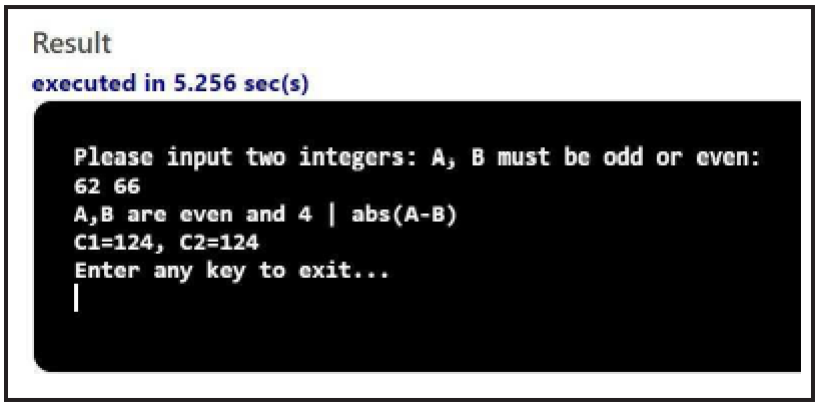

Figure 4. Testing the bit-wise XOR operation by Language Python.

The authors give an example by Equation (14) and provide some proofs such as Lemma 1 to Lemma 4, and Theorem 1, we show the experiment result in Figure 4. The source code by Language Python in listing 2, namely 'TEST-XOR.py'. The reader can copy and paste on online compiler such as "https://www.jdoodle.com" to check the result.

Listing 2. TEST-XOR.py

```
while True:
    A, B = map(int, list(input('Please input two integers: A, B must be odd or even:\n').split(" ")))
    if (A % 2 == 0 and B % 2 != 0) or (A % 2 != 0 and B % 2 == 0):
        print("error:A, B must be odd or even numbers\n")
    elif A % 2 != 0 and B % 2 != 0:
        print("A, B are odd numbers")
        c1 = A ^ B
        c2 = (-A) ^ (-B)
        print("C1=%d, C2=%d" % (c1, c2))
        break
    else:
        # If A, B are even numbers
        if A % 4 == 0 and B % 4 == 0:
            if A % 8 != 0 and B % 8 != 0:
                # A, B are divisible by 4 rather than 8
                print("A, B be divided 4 and be not divided 8")
                c1 = A ^ B
                c2 = (-A) ^ (-B)
                print("C1=%d, C2=%d" % (c1, c2))
                break
            else:
                print("a or b is divisible by 8\n")
        else:
            if A % 4 == B % 4:
                print("A,B are even and 4 | abs(A-B)")
                c1 = A ^ B
                c2 = (-A) ^ (-B)
                print("C1=%d, C2=%d" % (c1, c2))
                break
            else:
                print(" A,B are even but A or B or abs(A-B) is not divisible by 4\n")
print('Enter any key to exit...'
```

V. Conclusion

In this paper the authors describe the XOR expressions. Formal verification does not guarantee 100% whether there are errors in logical inferences (such as tautology). At least in the process stage or the result stage of inference, it plays an important decisive role. It is very hard to find or check the problem from mathematics or informatics fields. The authors fully present this situation by Galois field 2-adic number.

References

[1] J. Zhang and Y. Wang, "An improved signature scheme without using one-way hash functions," Applied Mathematics and Computation, vol. 170, no. 2, pp. 905–908, November 2005.

[2] F.-Y. Yang and M.-H. Hsu, "Security analysis Zhang-Wang signature scheme without using one-way hash functions," in International Conference on Accounting and Information Technology.

[3] L.-F. Guo, Y. Li, and L. Hu, "Cryptanalysis of a signature scheme without using one-way hash functions," Journal of the Graduate School of the Chinese Academy of Sciences, vol. 25, no. 5, pp. 698–700, 2008.

[4] C. Liu and J. Zhang, "Security analysis of Zhang-Wang digital signature scheme," Communications of the CCISA, vol. 15, no. 4, pp. 24–29, 2009.

[5] J. Zheng, "Security analysis of Boolean algebra based on Zhang-Wang digital signature scheme," in AIP Conference Proceedings, vol. 1618, pp. 507–509, 2014.

[6] Y.-Z. Luo, C. Zhang, C. Liu, and C.-W. Hsu, “Algebraic attack on Zhang-Wang digital signature algorithm,” in AIP Conference Proceedings, pp. 941–943, 2019.
[7] H. C. A. van Tilborg, Encyclopedia of Cryptography and Security, H. C. A. van Tilborg, Ed. Springer, 2005.
[8] Wikipedia, “Algebraic attack,” Website, http://en.citizendium.org/wiki/Algebraic attack, 2018.
[9] L. Xu, C. Liu, and N. Wang, “Comment on an improved signature without using one-way hash functions,” in 2010 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, pp. 441–443, 2010.
[10] W. Yang and C. Liu, “Analysis of parameters probability on Zhang-Wang signature scheme,” in AIP Conference Proceedings, vol. 1618, pp. 510–512, 2014.
[11] J. Qui, G. Chen, S. Cheng, Y. Ou, and C. Liu, “Probability analysis for Zhang-Wang signature scheme,” in AIP Conference Proceedings, pp. 141–143, 2018.
[12] X.-L. Luo, C. Liu, and S. C.-I. Chen, “The myth of disassembly and reverse engineering based on a digital signature scheme,” Design Engineering, no. 2, pp. 341–353, 2021.
[13] C. Yu and M. Ciesielski, “Formal analysis of Galois field arithmetic circuits-parallel verification and reverse engineering,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 2, pp. 354–365, 2019.
[14] C. Liu, S. Chen, and S. Sun, “Security of analysis mutual authentication and key exchange for low power wireless communications,” Energy Procedia, vol. 17, pp. 644–649, 2012.
[15] J. Fang, C. Liu, and J. Wu, J. C. Hung, N. Y. Yen, and K.-C. Li, “Weakness of an Elgamal-like cryptosystem for enciphering large messages,” in Frontier Computing, Eds, pp. 1225–1231, 2016.
[16] J. Zheng, “Software code security analysis based on assembler instruction decoding mappings,” Journal of Longyan University, vol. 32, no. 2, pp. 35–42, April 2014.
[17] A. Mohanta and A. Saldanha, Debuggers and Assembly Language. Berkeley, CA: Apress, pp. 525–638, 2020.