Εργασία Ψηφιακές Επικοινωνίες - CRC

Ονοματεπώνυμο: Ασημάκης Κύδρος
ΑΕΜ: 3881
asimakis@csd.auth.gr

10 Ιουνίου 2022

0 Εισαγωγή

Θα υλοποιήσουμε την διαδικασία ανίχνευσης σφαλμάτων σε σήματα μέσω κυκλικών κωδικών.

1 Κώδικας

Υιοθετούμε τα strings για να περιγράψουμε την έννοια των σημάτων. Θα χρειαστεί επομένως να ορίσουμε την έννοια της πρόσθεσης XOR και της διαίρεσης modulo-2.

Εφόσον χρησιμοποιείται στην διαίρεση, θα ξεκινήσουμε με τη **πρόσθεση XOR**:

```
string add_xor(const string &bigger, const string &smaller){
    int difference = (int)bigger.length() - (int)smaller.length(), i = 0;
    //make sure the bigger parameter is first
    if (difference < 0) return add_xor( bigger: smaller,  smaller: bigger);

    string sum;
    for(char bit : bigger)
        //bigger signal's starting bits will not participate in the addition
        if(difference-- > 0) sum += bit;
        else sum += (bit == smaller[i++]) ? '0' : '1';
    return sum;
}
```

Παρατηρούμε πως τα σήματα που λαμβάνουμε μπορεί να μην είναι ισομήκη. Αυτό το πρόβλημα πρέπει να λυθεί, καθώς πρέπει να προσθέτουμε αντίστοιχα bits κάθε φορά.

Κάνουμε την παραδοχή πως το πρώτο όρισμα θα είναι τουλάχιστον ίσο με το δεύτερο. Αν όχι, επιστρέφουμε αναδρομικά το άθροισμα με ανταλλαγμένα τα ορίσματα. Έχοντας πλέον το μεγαλύτερο, μετράμε την διαφορά τους και εισάγουμε στο sum αυτούσια τα πρώτα bits του μεγαλυτέρου, τόσα όσα είναι η διαφορά, αφού δεν θα συμμετάσχουν σε κάποια πρόσθεση (η XOR δεν αφήνει υπόλοιπα).
Να σημειωθεί πως, αν τα σήματα τελικά ήταν ισομήκη, τότε απλά δεν προσθέτουμε τίποτα.

Εξασφαλίσαμε ότι τα υπολοιπόμενα σήματα έχουν ίσο μήκος. Εκτελούμε την XOR πρόσθεση

$$\forall(bit1, bit2) : XOR(bit1, bit2) = \begin{cases} 0, & \text{αν } bit1 = bit2 \\ 1, & \text{αλλιώς} \end{cases}$$

και εισάγουμε το αποτέλεσμα στο τέλος του sum.

Έχοντας την υλοποίηση της πρόσθεσης μπορούμε να ορίσουμε πλέον και την **διαίρεση mod-2:**

```
string division_mod2(const string &dividend, const string &divisor){
    int length = (int)divisor.length(), end = length;
    string remainder;

    for(char bit: dividend)
        if (length-- > 0) remainder += bit; //remainder initialization
        else{
            //perform addition only if remainder is normalized
            if (remainder.front() != '0')
                remainder = add_xor( bigger: remainder,  smaller: divisor);
            for(int i = 1; i < end; i++) //shift left by one
                remainder[i - 1] = remainder[i];
            remainder[end - 1] = bit; //add the next bit of dividend
        }
    for(char bit : remainder) if (bit != '0') return remainder;
    //if remainder describes decimal zero, return only 0
    return "0";
}
```

Ξεκινάμε αρχικοποιώντας το υπόλοιπο (remainder) στα n αρχικά bits του διαιρετέου, όπου n το μήκος του διαιρέτη. Σε κάθε κύκλο πρέπει, προτού γίνεται η πρόσθεση του υπολοίπου με τον διαιρέτη, να εξασφαλίζουμε πως το υπόλοιπο δεν έχει παραπανήσια μηδενικά στην αρχή του, κάνοντας shift left κατά ένα άν το MSB είναι μηδέν.

Επειδή ο διαιρέτης ξεκινά εξόρισμού με 1, το αποτέλεσμα της πρόσθεσης (το καινούριο remainder) θα ξεκινά σίγουρα

με 0, άρα θέλουμε και πάλι ένα shift left by one. Στο τέλος κάθε κύκλου εισάγουμε στο υπόλοιπο το επόμενο bit του διαιρετέου.

Προτού επιστρέψουμε το τελικό αποτέλεσμα, το σαρώνουμε μια φορά για να ελέγξουμε αν ορίζει το δεκαδικό μηδέν. Σε περίπτωση που το ορίζει, επιστρέφουμε απλά μηδέν, για σαφήνεια.

Μπορούμε τώρα να προσομοιώσουμε τον έλεγχο. Ακολουθώντας τον ορισμό της θεωρίας καταλήγουμε στο παρακάτω:

```
std::mt19937 engine( sd: time( Time: nullptr) + 1);
std::uniform_int_distribution<int> uid( a: 0,  b: 1);

for(int i = 0, length = (int)P.length(); i < TESTS; i++){
    string D;
    //generate random D
    for(int j = 0; j < k; j++) D += std::to_string( val: uid( &: engine));
    //shift left by n - k == len(P) - 1
    for(int j = 1; j < length; j++) D += '0';
    //create T as instructed
    string T = add_xor( bigger: D,  smaller: division_mod2( dividend: D,  divisor: P));
    string dirty_T = noise_channel( clean_string: T,  bit_error_rate: BER, engine);
    if (T != dirty_T){ //if altered
        alterations++;
        if (division_mod2( dividend: dirty_T,  divisor: P) != "0") detections++;
    }
}
```

Κάθε φορά, δημιουργούμε ένα τυχαίο D . Έκτελούμε την γνωστή πράξη

$$T = 2^{n-k}D + 2^{n-k}DmodP$$

και ”περνάμε” το αποτέλεσμα T μέσα από ένα θορυβώδες κανάλι. Αν η είσοδος και η έξοδος του είναι διαφορετικές,

ξέρουμε ότι υπήρξε ψεγάδιασμα, και ελέγχουμε το υπόλοιπο R:

$$R = TmodP == 0?$$

για να αντλήσουμε τα ανάλογα αποτελέσματα.

2 Αποτελέσματα

Τρέχουμε την παραπάνω διαδικασία για τις αρχικές τιμές k = 20, P = "110101", BER = 10^{-3}:
1) πλήθος 10 εκατομμυρίων δειγμάτων:

```
Give k, P, BER:
20
110101
0.001
Wait...
Results from 10000000 random cases:
   -> Signals altered: 2.57563%
   -> Alterations detected: 99.9581%
   -> Alterations missed: 0.0419315%

Process finished with exit code 0
```

2) πλήθος 100 εκατομμυρίων δειγμάτων:

```
Give k, P, BER:
20
110101
0.001
Wait...
Results from 100000000 random cases:
   -> Signals altered: 2.56835%
   -> Alterations detected: 99.9614%
   -> Alterations missed: 0.0385851%

Process finished with exit code 0
```

3) πλήθος 1 δισεκατομμυρίων δειγμάτων:

```
Give k, P, BER:
20
110101
0.001
Wait...
Results from 1000000000 random cases:
   -> Signals altered: 2.56752%
   -> Alterations detected: 99.9618%
   -> Alterations missed: 0.0382275%

Process finished with exit code 0
```

Γίνεται εμφανές λοιπόν ότι η διαδικασία CRC είναι βέλτιστη και έγκυρη, καθώς η πιθανότητα να μην εντοπιστεί μια αλλοίωση είναι μηδαμινή.