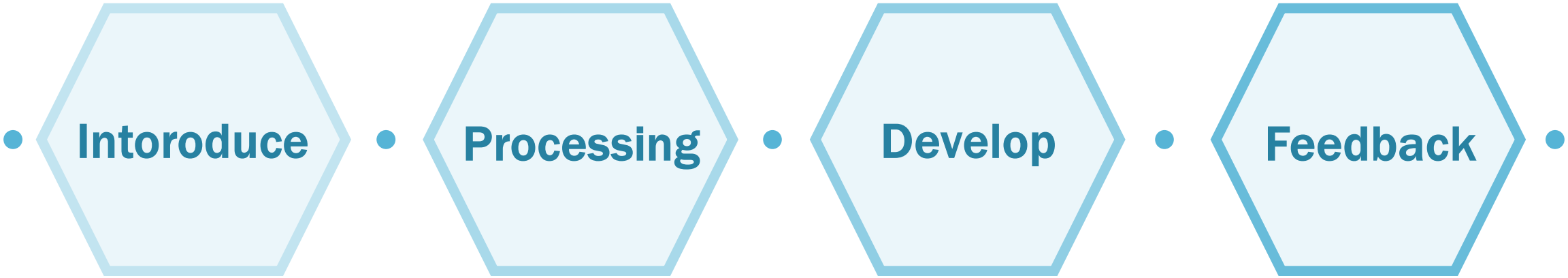
UBI Encryption: 우비 암호화

안성현 (Skiddie Ahn)

Contents



프로그램 소개

우비 암호화란?

암호화 과정

인코딩& 디코딩 과정

프로그램 개발

사용한 웹 서버 및
챗봇 API

피드백 및 업데이트

피드백 & 단점 극복

Introduce

프로그램 소개

우리만의 비밀코드: 우비 암호화

우비 암호화는 **RSA** 기반 암호화에 **UBI(우비)** 컨셉을 결합해서 **U,B,I** 조합의 암호를 만드는 것을 말한다.

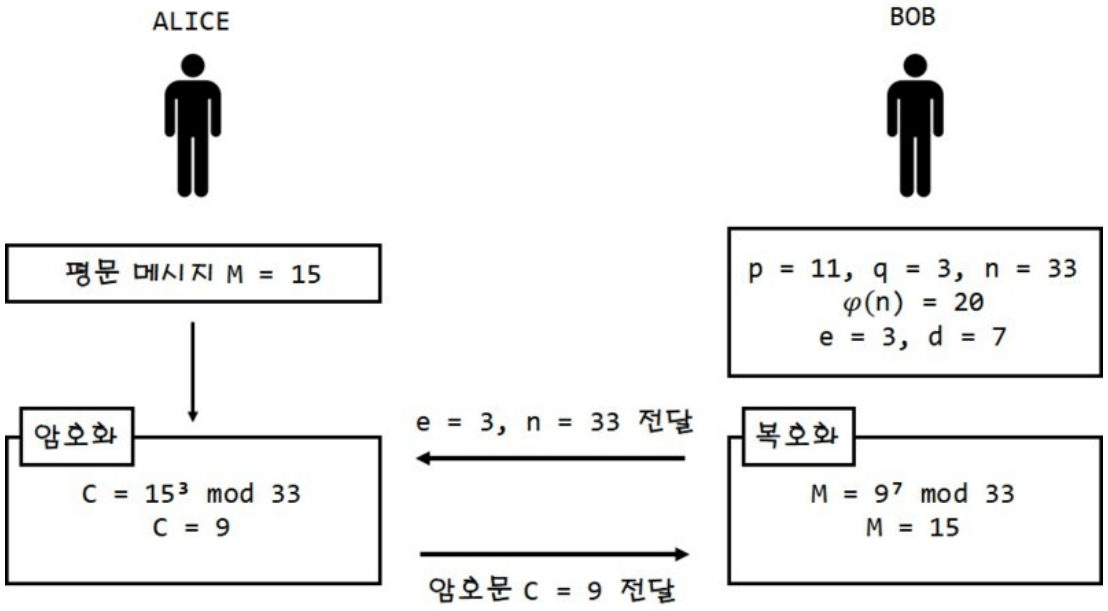
암호라는 것이 이미지가 딱딱해서 중요성을 모르는 사람들이 많다고 생각한다.

따라서 '우비'같이 비를 연상케 하는 친근한 이미지와 편리하게 암호를 만들 수 있는 시스템이 있으면 개선 될 것이라고 기대하였다.

또한 기존에 암호문을 만들어 보려 했으나, 복잡해서 포기한 분들에게도 도움이 될 것이라고 생각한다.

Introduce

RSA 방식과 UBI 방식



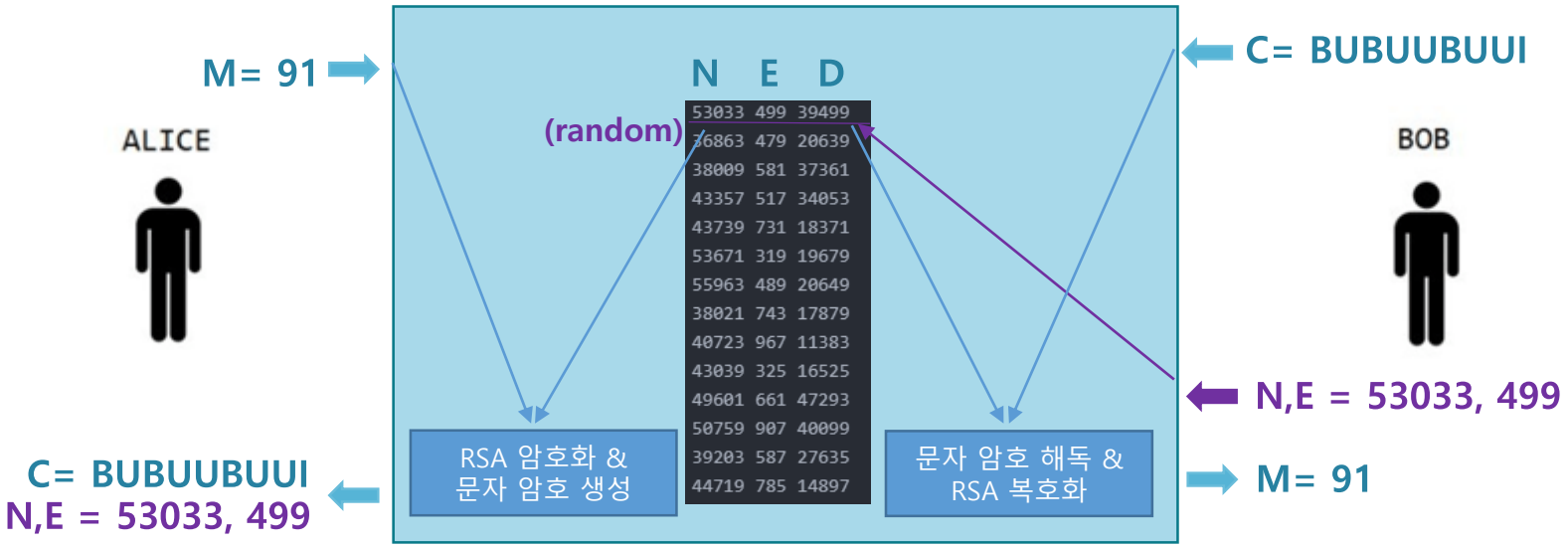
d값(개인 키) : e mod(p-1)(q-1)의 역
n값(공개 키) : p x q (단, p, q는 소수)
e값(공개 키) : (p-1)*(q-1)과 서로소 (1 < e < n)
암호화 공식 : M이 메시지일 때, M^e mod n = C (0 <= M <=n-1)
복호화 공식 : C가 암호문일 때, C^d mod n = M
참고 : (**서로소** : 1이외의 공약수가 없는 관계)

RSA는 현재 가장 많이 사용되는 공개키 암호화 알고리즘이다.
두 사람이 공유하는 공개 키(n,e)가 있고 복호화에 필요한 개인키(d)가 있는데 이 키와 공식을 이용해서 암호화 및 복호화를 진행한다.

예를 들어 A가 '15'라는 메시지를 공개키와 암호화 공식으로 '9'라고 바꿔서 전달하면 'B'가 개인키와 복호화 공식으로 '15'라고 해독할 수 있다.

Introduce

RSA 방식과 UBI 방식



UBI는 RSA와 다음과 같은 차이를 지닌다.

1. 암호화 시 공개키를 입력하지 않는다. (프로그램이 랜덤 생성)
2. 복호화 시 전달 받은 공개키(n,e)를 이용한다. (프로그램 내부에서 개인키(d)를 가져옴)
3. 프로그램에 등록된 키(n,e,d)만 인식한다. 따라서 복호화 시 입력한 공개 키가 리스트에 있는지 확인하는 작업이 있다.
4. U,B,I 조합의 문자 암호를 생성하고 해독하는 과정이 있다.
5. 차별화를 주기 위해 공개키(e,n)는 우비코드(e,n)로 부르기도 한다.

Introduce

C ProtoType & JS Module

```
■ 암호화할 문장을 적어 주세요:
hello
(총 5자)
■ 다음과 같이 암호화되었습니다:
BBUUUBBUBUBUUIBUBBUUUUUBUUUBUIBUBBBBBUUBUUUUBUI
(총 47자)
[ 우코드(e), 비코드(n) ] = [ 557, 66757 ]
```

```
■ 우코드(e)를 입력해주세요:557
■ 비코드(n)를 입력해주세요:66757
■ 복호화할 문장을 적어 주세요:
BBUUUBBUBUBUUIBUBBUUUUUBUUUBUIBUBBBBBUUBUUUUBUI
(총 47자)
■ 다음과 같이 복호화되었습니다:
hello
(총 5자)
```

C언어로 미리 제작된 ProtoType이다.
기본적으로 영문과 숫자, 특수기호를 암호화 할 수 있다.

```
1  const UBI=require('ubicrypt');
2
3  let string="Node.js 우비 암호화 Test!!"
4  console.log('원문:',string)
5
6  let Eobject=UBI.Encryption(string);
7  console.log('암호문:',Eobject.string);
8  let Dobject=UBI.Decryption(Eobject.string);
9  console.log('복호문:',Dobject.string);
```

```
Atom Runner: test.js

원문: Node.js 우비 암호화 Test!!
암호문: UBUUUIUIIUBUIUBUBBUIBUUBUIBIUUUIUIIUIIUBIUUIIIUIIUUUIBBUUUIUUIUUIUUUUUBUUBUIUBBIIBUIIIUBIBBBUUBUIUUIBIBBBIUIUBBUBBUBIB
복호문: Node.js 우비 암호화 Test!!

Running: node (cwd=C:₩Users₩tjdgu₩OneDrive₩바탕 화면₩test.js pid=6452). Exited with code=0 in 0.086 seconds.
```

암호화를 쉽게 하기 위해 JS 모듈을 만들어보았다.
'npm i ubicrypt'명령어로 다운로드할 수 있다.
한글,영문,기호를 암호화 할 수 있고 우비코드 없이 복호화가 가능하다.

Processing

Encoding (I)

```
Node.js 우비 암호화 Test!!
Node.js `우비` `암호화` Test!!
Node.js `dnql` `dkaghghk` Test!!
```

```
[
  40, 15,  4,  5, 65, 10, 19, 63, 92,
   4, 14, 17, 12, 92, 63, 92,  4, 11,
   1,  7,  8,  7,  8, 11, 92, 63, 46,
   5, 19, 20, 70, 70
]
```

```
[
  4015,  405, 6510, 1963,
  9204, 1417, 1292, 6392,
   411,  107,  807,  811,
  9263, 4605, 1920, 7070
]
```

1. 입력된 문자열에서 한글을 찾고 ``로 구분 짓는다.
이후 ``안쪽 문자열을 Inko 모듈의 ko2en함수를 이용해서 영어로 변경한다.

- 원 문자열(input) -> 영어 처리
- ex) 우비 -> `우비` -> `dbql`

2. 변경된 문자열을 한 글자씩 숫자로 변경해서 '전처리 코드(code)' 배열에 저장한다.

- 문자 -> 전처리 코드(code) ex) a->1, b->2, ... , A->27, B->28, ...

3. '전처리 코드(code)'를 두 개씩 묶어서 '원 숫자(bundle)' 배열에 저장한다. 만일 code가 홀수개면 마지막 code는 100을 곱해서 저장한다.

- 전처리 코드(code) -> 원 숫자(bundle) ex) 40,15 -> 4015

Processing

Encoding (II)

```
[
  40527, 19476,  6510,
  10765, 24526, 42493,
  29491,  1502, 16385,
  41183, 11076, 16133,
  29801,  6072, 23273,
  32335
]
```

```
[
  '1200220102201',   '0200112222101',
  '21200200201',     '0200222212001',
  '101220102101',    '1210120011101',
  '10001120010201',  '02200021001',
  '0221110002001',   '1202020001101',
  '0202002222101',   '022102111101',
  '100010000020201', '21002221001',
  '101110221101',    '111011112001'
]
```

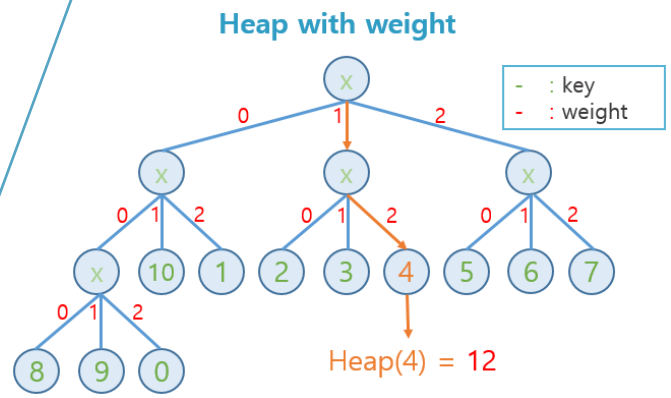
4. 원 숫자에 RSA 암호화를 적용해서 '암호 숫자(encode)' 배열에 저장한다.

- 원 숫자(bundle) -> 암호 숫자(encode) ex) 4015 -> 40527
- 우코드(e)=487 , 비코드(n)= 46129
- C = M^e mod n = 4015^487 mod 46129 = 40527

5. 오른쪽과 같은 3-Heap이 있다고 가정하자. 이 Heap은 leaf node 만 의미 있는 정보를 지닌다. 또한 트리의 가지에는 weight가 부여됐다. 이 때, leaf의 key값은 root에서 해당 leaf까지의 weight를 결합한 것으로 변환이 가능하다.
예를 들어 4는 concat('1', '2')='12'로 변환이 가능하다.
이제 encode의 숫자를 leaf의 key값으로 생각하고 각 원소에 대해 위 로직을 적용한다. 적용한 결과는 '힙 숫자(hcode)'배열에 저장한다.

- 암호 숫자(encode) -> 힙 숫자(hcode) ex) 40527 -> '1200220102201'

힙 숫자는 항상 (leaf:10)에 해당하는 weight 결합으로 마무리를 짓는다. (for. Decoding)




```
IBUUBBUIUBBUIUBUUIIBBBBIUIBIBUUBUUBUIUBUUBBBBIBUUIIUIBBUIUBI
```

6. 힙 숫자들을 모두 결합한 뒤 U,B,I 문자들로 변경한다. (0->'U', 1->'I', 2->'B')

- 힙 숫자(hcode) -> 암호 문자열(Estring) ex) '1200....' -> 'IBUU....'

Processing

Decoding (I)

```
IBUUBBUIUBBUIUBUUIIBBBBIUIBIBUUBUUBUIUBUUBBBBIBUUIIUIBBUIUBI
```

1. 암호 문자열(Estring)을 숫자들로 변경한다. (‘U’->0, ‘I’->1, ‘B’->2)
이 숫자들을 ‘복호화 전처리 코드(code2)’라고 부른다.

- 암호 문자열(Estring) -> 복호화 전처리 코드(code2) ex) ‘IBUU..’ -> ‘1200..’

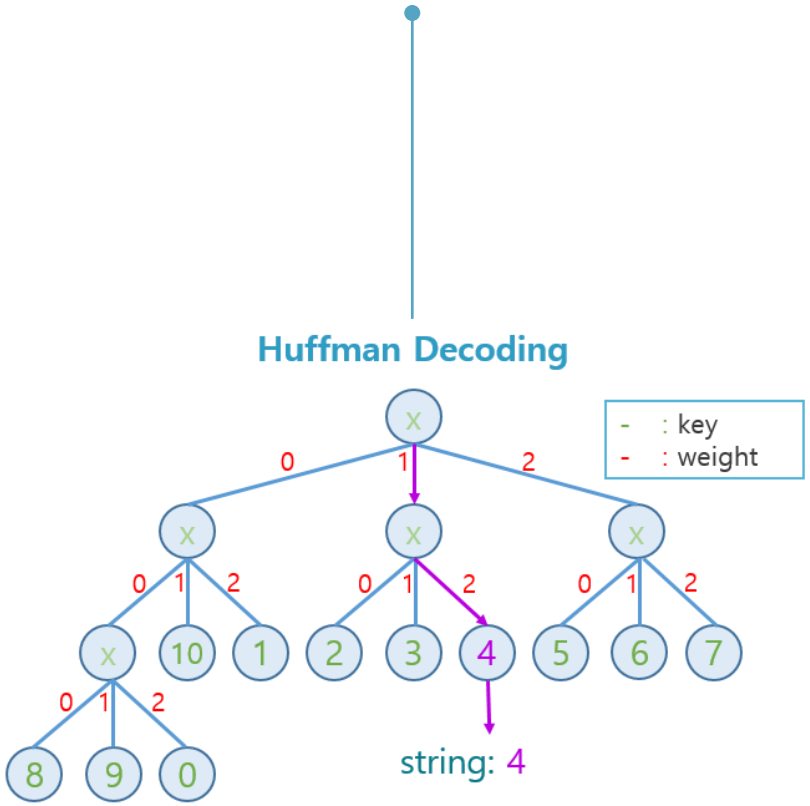
```
1200220102201020011222210121200200201020022221200110122010210
```

```
[
  40527, 19476,  6510,
  10765, 24526, 42493,
  29491,  1502, 16385,
  41183, 11076, 16133,
  29801,  6072, 23273,
  32335
]
```

2. code2를 Huffman Decoding한다.
‘허프만 부호화’의 Decoding 부분을 생각하면 된다.
이 때 허프만 트리는 인코딩할 때 사용한 3-heap으로 대체한다.
디코딩 결과는 string이라는 임시 문자열에 결합하면서 저장되는데
만약 결과가 10이 나온다면 string을 정수형으로 변경하고
‘변환 숫자(c_code)’배열에 저장한다.

- 복호화 전처리 코드(code2) -> 변환 숫자(c_code)
- ex) ‘1200..’ -> [40527, 19476, ...]

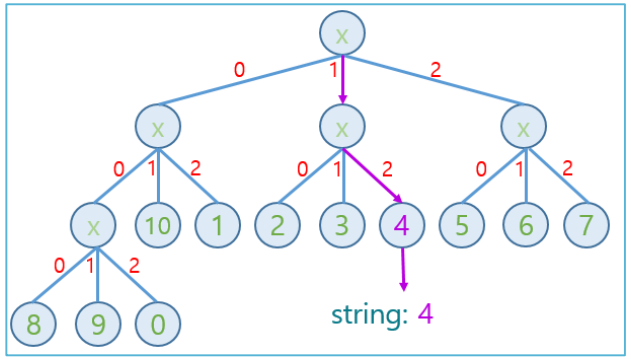
허프만 부호화는 문자의 빈도수를 기준으로 문자열을 압축하는 기법이다. 원래 문자열을 기준으로 각각 다른 허프만 트리를 형성하지만, 암호화에 이용해야 하므로 허프만 트리를 고정시키고 Huffman Decoding 부분만 이용하였다. 그래도 leaf의 키값들은 필자가 임의로 지정하므로 역추적이 쉽지 않다. (역추적 경우의 수: 11!)



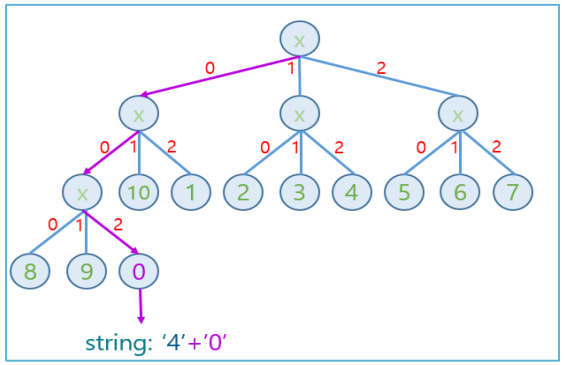
Processing

Input: 0 -> node = node->left
Input: 1 -> node = node->center
Input: 2 -> node = node->right
leaf node -> string += leaf node.key, node = root
leaf node(10) -> c_code.add(string), init(string)

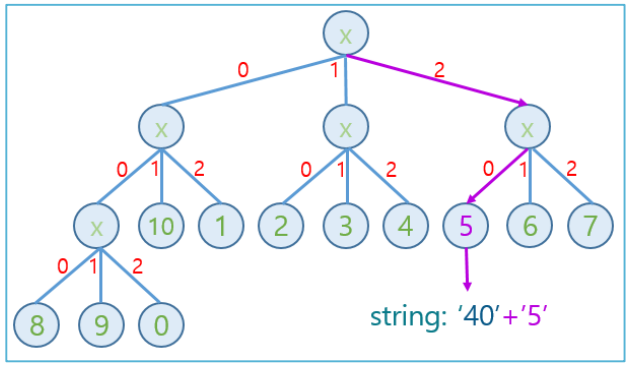
Huffman_decoding(1200220102201…)



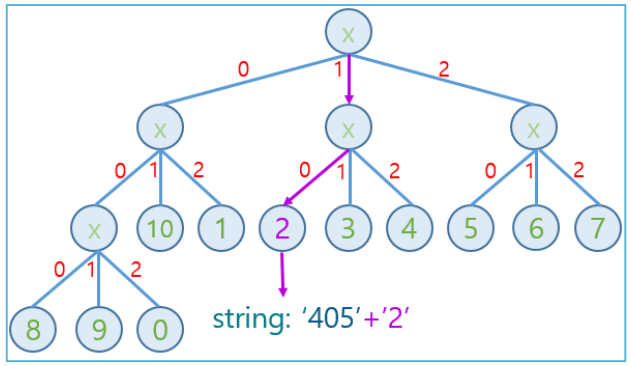
12/



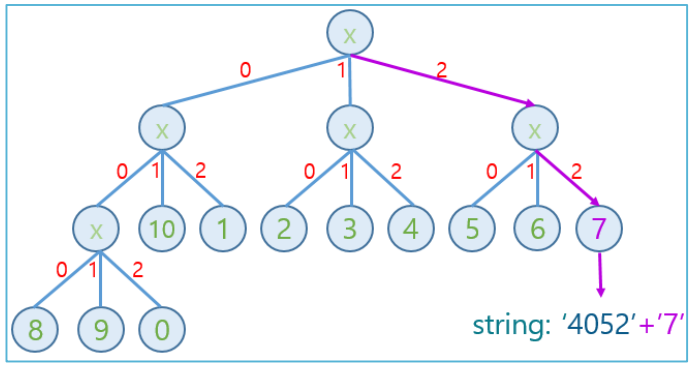
12/002/



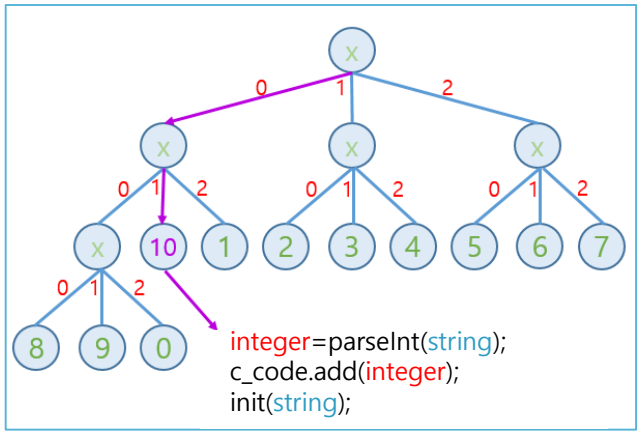
12/002/20/



12/002/20/10/



12/002/20/10/22



12/002/20/10/22/01

Processing

Decoding (II)

```
[
  4015,  405, 6510, 1963,
  9204, 1417, 1292, 6392,
   411,  107,  807,  811,
  9263, 4605, 1920, 7070
]
```

```
[
  40, 15,  4,  5, 65, 10, 19, 63, 92,
   4, 14, 17, 12, 92, 63, 92,  4, 11,
   1,  7,  8,  7,  8, 11, 92, 63, 46,
   5, 19, 20, 70, 70
]
```

```
Node.js `dnql` `dkaghghk` Test!!
Node.js `우비` `암호화` Test!!
Node.js 우비 암호화 Test!!
```

3. 변환 숫자에 RSA 복호화를 적용해서 '복호 숫자(decode)'배열에 저장한다.

- 변환 숫자(c_code) -> 복호 숫자(decode) ex) 40527 -> 4015
- 비코드(n)= 46129, d= 5347
- M = C^d mod n = 40527^5347 mod 46129 = 4015

4. 복호 숫자를 두 부분으로 분할해서 '후처리 코드(l_code)'배열에 저장한다.

- 복호 숫자(decode) -> 후처리 코드(l_code) ex) 4015 -> 40,15

5. 후처리 코드를 문자로 바꾼 다음 문자열로 합친다. 이후 ``안쪽 문자열을 Inko 모듈의 en2ko함수를 이용해서 한글로 변경한다.

- 후처리 코드(l_code) -> 복호 문자열(Dstring) -> 한글 처리
- ex) 92,4,14,17,12,92 -> `dbql` -> `우비` -> 우비

Processing

All Process

```
Node.js 우비 암호화 Test!!
Node.js `우비` `암호화` Test!!
Node.js `dnql` `dkaghghk` Test!!
[
  40, 15,  4,  5, 65, 10, 19, 63, 92,
   4, 14, 17, 12, 92, 63, 92,  4, 11,
   1,  7,  8,  7,  8, 11, 92, 63, 46,
   5, 19, 20, 70, 70
]
[
  4015,  405, 6510, 1963,
  9204, 1417, 1292, 6392,
   411,  107,  807,  811,
  9263, 4605, 1920, 7070
]
[
  40527, 19476,  6510,
  10765, 24526, 42493,
  29491,  1502, 16385,
  41183, 11076, 16133,
  29801,  6072, 23273,
  32335
]
[
  '1200220102201',   '0200112222101',
  '21200200201',     '0200222212001',
  '101220102101',    '1210120011101',
  '10001120010201',  '02200021001',
  '0221110002001',   '1202020001101',
  '0202002222101',   '022102111101',
  '100010000020201', '21002221001',
  '101110221101',    '111011112001'
]
IBUUBBUIUBBUIUBUUIIBBBBIUIBIBUUBUUBUIUBUUBBBBIBUUIIUIBBUIUBIUIIBIUIBUUIIIUIIUUUIIBUUI
```

원 문자열(input)
-> 한글 처리

전처리 코드(code)

원 숫자(bundle)

암호 숫자(encode)

힙 숫자(hcode)

암호 문자열(Estring)

Encoding

```
IBUUBBUIUBBUIUBUUIIBBBBIUIBIBUUBUUBUIUBUUBBBBIBUUIIUIBBUIUBIUIIBIUIBUUIIIUIIUUUIIBUUI(
1200220102201020011222210121200200201020022221200110122010210112101200111011000112001(
[
  40527, 19476,  6510,
  10765, 24526, 42493,
  29491,  1502, 16385,
  41183, 11076, 16133,
  29801,  6072, 23273,
  32335
]
[
  4015,  405, 6510, 1963,
  9204, 1417, 1292, 6392,
   411,  107,  807,  811,
  9263, 4605, 1920, 7070
]
[
  40, 15,  4,  5, 65, 10, 19, 63, 92,
   4, 14, 17, 12, 92, 63, 92,  4, 11,
   1,  7,  8,  7,  8, 11, 92, 63, 46,
   5, 19, 20, 70, 70
]
Node.js `dnql` `dkaghghk` Test!!
Node.js `우비` `암호화` Test!!
Node.js 우비 암호화 Test!!
```

복호화 전처리 코드(code2)

변환 숫자(c_code)

복호 숫자(decode)

후처리 코드(l_code)

복호 문자열(Dstring)
-> 한글 처리

Decoding

Develop

Kakao Bot API

Kakao business에서 심사를 통과 받고 API를 제공받았다.
Bot이 정적인 대화가 아닌 동적인 대화를 해야 되기 때문에
(풀백 블록-스킬 데이터 사용)을 이용하였다.

'스킬 데이터'는 '사용자가 작성한 코드'를 의미하며
모든 코드는 Express 프레임워크를 이용해서 작성하였다.
해당 프레임워크에 대한 지식은 '생활코딩'으로부터 얻었다.

Amazon AWS EC2

Bot이 요청에 응답하기 위한 웹 서버는 Amazon AWS에서 제공하는 프리티어 Ubuntu(18.04 LTS)를 이용하였다.

Intel Xeon CPU와 1GB 메모리를 제공받았지만
테스트 결과 준수한 성능을 보였다.

웹 서버는 사용자 증가에 따라 변경할 예정이다.

Feedback

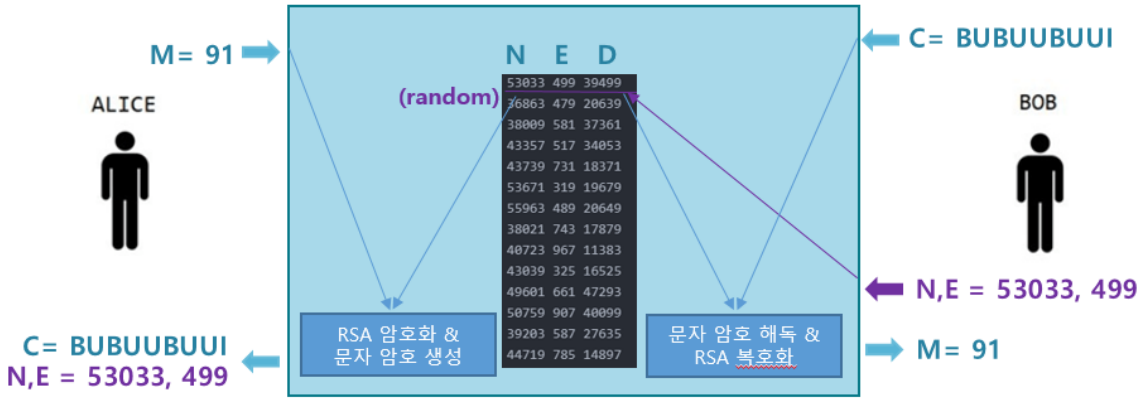
UBI 방식의 장/단점

장점(Pros)

- 암호화 시 공개키를 입력하지 않아 암호를 만들기가 편리하다.
- 기존의 복잡한 암호와는 달리 **U,B,I** 형태의 간단한 암호이기에 거부감이 줄어든다.
- 암호를 만들기 전 공개키를 보내고 다시 해독을 하기 위해 개인키를 찾아야 하는 번잡한 과정이 없다.
- **Key**값이 작고 효율적인 알고리즘으로 암호화 및 복호화 속도가 빠르다.
- 문자열 처리 및 디코딩 알고리즘이 추가됨으로써 **UBI Encryption** 프로그램이 없으면 해독이 어렵다.
- **Ubicrypt** 모듈을 사용할 경우, 개발자가 원하는 **Key**값을 등록할 수 있다.

단점(Cons)

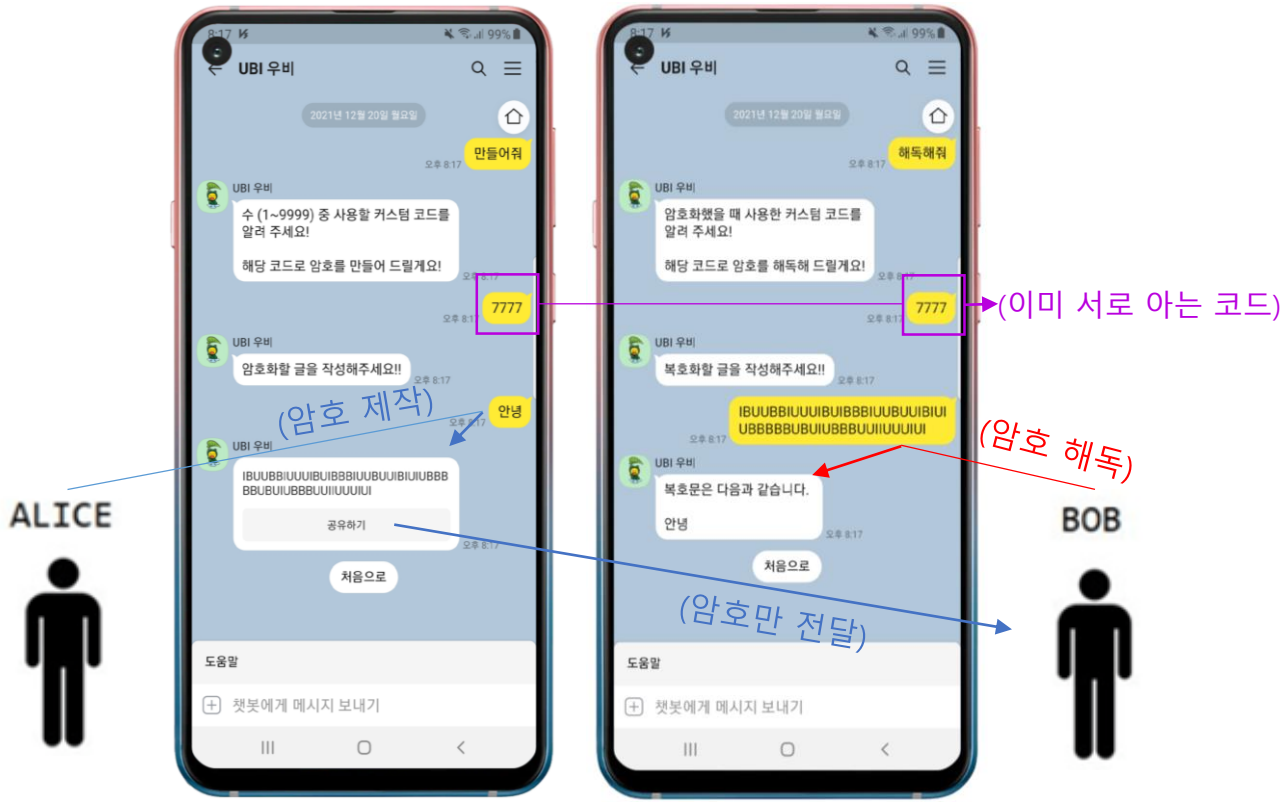
- 항상 키 값이 랜덤으로 생성돼서 키 공유가 번거롭다.
- 여러 문자를 **U,B,I** 세 문자로 표현하다 보니 암호가 길어진다.
- 키 값이 작아 기존 암호화 방식에 비해 안정성이 떨어진다.
- 공개키가 외부에 공유되고 제3자가 **UBI**방식을 알고 있다면 정보 유출 가능성이 있다.



Feedback

단점 극복

- 공개키가 외부에 공유될 경우 정보 유출 가능성이 있다. -> 해결
- 항상 키 값이 랜덤으로 생성돼서 키 공유가 번거롭다. -> 해결



< 커스텀 모드 >

커스텀 모드에서는 사용자가 지정한 코드로 암호화와 복호화가 가능하다.

만약 **A**와 **B**가 '**7777**'이라는 코드를 사용하자고 약속하였다.
그리고 **A**가 **B**에게 보내기 위한 암호를 만들었다.
이제 **A**는 키를 제외한 암호만 보내면 된다.
B는 암호를 받고 '**7777**'코드로 해독이 가능하다!

-> 커스텀 모드를 사용할 시 공개키라는 개념이 없어진다.
따라서 공개키 때문에 정보가 유출될 일이 없다.

-> 서로가 알고 있는 키로 암호화와 복호화를 할 수 있어서 **UBI**의 장점이자 단점인 '키 랜덤 생성'문제를 해결한다.

감사합니다