Implementation And Analysis Of A Polyalphabetic Substitution Cipher Including Attacks

Matthias Riegler <mriegler@htwsaar.de>, 3761312

Description Of The Cipher

The cipher presented in *A New Symmetric Key Encryption Algorithm Using Images as Secret Keys* encrypts arbitrary ASCII data and uses a given image as a shared secret. It substitutes the ASCII to ciphertext by mapping characters to pixel positions, thus generating a monoalphabetic substitution cipher.

Implementation

During this project, the described cipher was implemented using the Go programming language aiming at a high encryption and decryption performance.

Key Verification

As described in the underlying paper, the substitution alphabet is derived from a given image source. The pkg/image package implements an interface to read and write arbitrary images used as keys and provides an acceptance test checking the validity of using an image as a key for the cipher.

The cipher requires a map of the (7bits) ASCII alphabet to a group of pixel positions representing each character. The mapping shall be one-way invertible, meaning ’a <=> any of { 1, 2, 3, 4, 5 }. For an image to be accepted as a valid key, all ASCII characters must have at least one pixel representing the character. Allowing one-way inversion of character to pixel mapping, a pixel value is ANDed with0b01111111‘, thus eliminating the most significant bit. Once an image fulfills the requirement of mapping all ASCII characters to one or many pixels, it is accepted. For every character, the encryption chooses a random pixel representing the character.

```go
func CheckAccept(i *Image) bool {
    acceptanceMap := make(map[uint8]bool)
    for _, b := range i.Data {
        acceptanceMap[b&0b01111111] = true
    }

    // Check if ASCII alphabet can be represented
    for c := 0; c < 128; c++ {
        if _, ok := acceptanceMap[uint8(c)]; !ok {
            return false
        }
    }
    return true
}
```

For testing purposes, the function `Mock()` generates an image with 128 pixels width and 128 height containing random pixel values. This iamge can be verified to be a valid key for the cipher:

```go
// Mock creates an image with 128x128 dimension for testing purposes
func Mock() *Image {
    // Generate mock image
    i := &Image{
        Data:      make([]uint8, 128*128, 128*128),
        Dimension: Dimension{128, 128},
    }

    // Fill mock image with mock data
    for h := 0; h < i.Dimension.Height; h++ {
        for w := 0; w < i.Dimension.Width; w++ {
            i.Data[w+i.Dimension.Width*h] = uint8(rand.Intn(128))
        }
    }

    return i
}
```

Encryption

Before the encryption starts, the key is converted into an optimized data structure implemented by a hash-map with the character to be encrypted as a key and an array of pixel positions representing this character.

```go
// PixelPosition represents the position of a pixel on the image
type PixelPosition struct {
    Width  int `json:"width"`
    Height int `json:"height"`
}

// PixelGroups represents a grouping of pixels based on their value
type PixelGroups map[uint8][]PixelPosition
```

The key, an arbitrary image, is loaded using the aforementioned `pkg/image` package and checked for validity. After it passes the test, a trivial iteration across all pixels helps generating the `PixelGroups` datastructure.

```go
// Encrypt allows encryption of an arbitrary ASCII string
func (c *Container) Encrypt(s string) (Encrypted, error) {
    enc := make(Encrypted, len(s))
    rnd := make([]byte, 4)

    // Iterate over the input string, determine (random) pixel position
    for i, b := range []uint8(s) {
        if pixelGroup, ok := c.PixelGroups[b]; ok {
            // Get the number of available options for the pixel value
            availOptions := len(pixelGroup)
            // Choose a random position out of the pixel group
            d := int(binary.BigEndian.Uint32(rnd)) % availOptions

            ppos := pixelGroup[d]
            enc[i] = ppos
        }
    }

    return enc, nil
}
```

During the encryption process, a loop iterates across the plaintext and determines the substitution for every character using a random entry out of the array mapped for the character. ”‘go // Encrypt allows encryption of an arbitrary ASCII string func (c *Container) Encrypt(s string) (Encrypted, error) { enc := make(Encrypted, len(s))

```
// Iterate over the input string, determine (random) pixel position
for i, b := range []uint8(s) {
    if pixelGroup, ok := c.PixelGroups[b]; ok {
        // Get the number of available options for the pixel value
        availOptions := len(pixelGroup)
        // Choose a random position out of the pixel group
        ppos := pixelGroup[rand.Intn(availOptions)]
        enc[i] = ppos
    }
}

return enc, nil
```

}

```
### Decryption
The Decryption builds on a lookup function.
A loop iterates across the ciphertext and retrieves the pixel value at the position the ciph

"`go
// Decrypt allows decryption of an arbitrary encrypted ASCII string
func (c *Container) Decrypt(enc Encrypted) (string, error) {
    dec := make([]byte, len(enc))
    for i, ec := range enc {
        // Check if in boundaries
        if ec.Height < c.Image.Dimension.Height && ec.Width < c.Image.Dimension.Width {
            // Calculate pixel position in the slice
            arrayPos := ec.Width + c.Image.Dimension.Width*ec.Height
            // Retrieve Byte
            dec[i] = byte(c.Image.Data[arrayPos] & 0b01111111)
        } else {
            // Invalid pixel position
            return "", errors.New("Invalid pixel position")
        }
    }

    // Convert encrypted bytes to string & return
    return string(dec), nil
}
```

Input/Output operations

For the ease of analyzing the ciphertext with different tools, a JSON format has been chosen, which encodes an array of `PixelPosition`. ”‘go // Encrypted contains a slice of PixelPositions type Encrypted []PixelPosition

// Reads a ciphertext into an internal datastructure func Read(f *os.File) ([]PixelPosition, error) { var out []PixelPosition dec := json.NewDecoder(f) // … return out, nil }

// Encodes ciphertext as JSON func Write(f *os.File, in []PixelPosition) error { enc := json.NewEncoder(f) if err := enc.Encode(in); err != nil { return err } return nil }

~~~~
### Testing The Implementation

To ensure the implementation is working and can encrypt & afterward decrypt ciphertext with
As an initial step, an initialization function generates a mocked image that passes the acce

Several test cases check the implementation for the desired functionality. The `Test_Extract
The `TestContainer_Encrypt_Decrypt` case verifies encrypted plaintext can be reverted to the


### Performance Testing The Implementation

The performance of the encryption & decryption operations is implemented using the Go intern
Different test cases for parallel and synchronous chunk decryption and encryption are used t

Benchmark implementation:
```go
~~~~

```
func BenchmarkContainer_Encrypt1MByte(b *testing.B) {
    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            cipher.Encrypt(testData1MByte)
        }
    })
}

func BenchmarkContainer_Encrypt1024(b *testing.B) {
    for n := 0; n < b.N; n++ {
        cipher.Encrypt(testData1MByte)
    }
}

func BenchmarkContainer_Decrypt1024(b *testing.B) {
    for n := 0; n < b.N; n++ {
        cipher.Decrypt(testData1MByteEnc)
    }
}

func BenchmarkContainer_Decrypt1MByte(b *testing.B) {
    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            cipher.Decrypt(testData1MByteEnc)
        }
    })
}
```

Benchmark results (*Intel i7-9750H*):

```
xvzf in crypto/pkg/crypt on ? master $ go test -bench .
goos: darwin
goarch: amd64
pkg: github.com/xvzf/htw-crypto-project/pkg/crypt
BenchmarkContainer_Encrypt1Mbyte                      22      50381141 ns/op
BenchmarkContainer_Encrypt1Mbyte-6                    21      52016303 ns/op
BenchmarkContainer_Encrypt1Mbyte-12                   21      51928491 ns/op
BenchmarkContainer_Decrypt1Mbyte                     559       2031963 ns/op
BenchmarkContainer_Decrypt1Mbyte-6                   560       2022927 ns/op
BenchmarkContainer_Decrypt1Mbyte-12                  578       2020687 ns/op
BenchmarkContainer_Encrypt1MByteParallel              20      50529087 ns/op
BenchmarkContainer_Encrypt1MByteParallel-6            21     114059236 ns/op
BenchmarkContainer_Encrypt1MByteParallel-12           21     134402985 ns/op
BenchmarkContainer_Decrypt1MByteParallel             584       2014307 ns/op
BenchmarkContainer_Decrypt1MByteParallel-6          2061        527559 ns/op
BenchmarkContainer_Decrypt1MByteParallel-12         2544        456374 ns/op
PASS
```

```
ok      github.com/xvzf/htw-crypto-project/pkg/crypt    20.776s
```

Extrapolating the benchmark results of `BenchmarkContainer_Encrypt1Mbyte`, on an average of 21 runs, the encryption duration of 1 MB data is `50381141 ns`, which represents a throughput of *19.85 MB/s*. As seen in the results of the `BenchmarkContainer_Encrypt1MByteParallel-*` scenarios, running the encryption with multiple cores hurts the performance. This is likely due to the implementation and usage of the integrated random number generator.

However, the encryption benefits from running it in parallel and decrypts at `492 MB/s` with a single CPU core and `1.9 GB/s` with six cores. We are adding the hyper-threading of the Intel processor results in a further performance boost to `2.19 GB/s`.

Security Analysis

As described in the paper, the cipher maps one character to possibly multiple pixels in an image. While this seems to be secure, it is under some circumstances vulnerable to frequency analysis. Based on the plaintext length, the image dimensions, pixel distribution mapping to characters in the image, and the character frequency of the plaintext, it is possible to build groups of pixels based on their occurrence, assuming the random generator is uniform.

The security of the cipher, therefore, drastically depends on the plaintext length and the image size. Given a frequency vulnerable plaintext with a length of *10.000* characters containing *50* unique characters, the image used as key decides whether the cipher can be cracked or not: - The image contains distinguished pixels for all characters; thus, the ciphertext does not contain double values; **OTP like security** - The image does not contain distinguished pixels for all characters, resulting ciphertext values cannot be grouped based on their frequency; **Vulnerable to additional attacks, discussed in TODO** - The image does not contain distinguished pixels for all characters, resulting ciphertext values can be grouped based on their frequency; **Allows further analysis on simple substitution cipher**

Grouping Pixels

Given an image that allows pixel grouping, pixels can be grouped based on a similar occurrence. For this to work, a large ciphertext is required, depending on the image size used as a key. An optimized version of the k-means algorithm implemented by the python library kmeans1d has been used. In the first step, the frequency of each pixel in the ciphertext is analyzed. Afterward, the k-means is applied and used to build a map, reducing the problem space to a trivial substitution cipher. Testing showed this method works on small images (32x32 pixels) from time to time but unreliably. Other attack vectors are described in the following sections.

```python
def substitute(ciphertext, alphabet=ALPHABET):
    _total = len(ciphertext)

    _freq = {}

    for p in ciphertext:
        _freq[key(p)] = _freq.get(key(p), 0) + 1

    _sorted_freq = sorted(_freq.items(), key=operator.itemgetter(1), reverse=True)
    _freq_arr_val = [v for _, v in _sorted_freq]
    _freq_arr_key = [k for k, _ in _sorted_freq]

    # Cluster
    clusters, centroids = kmeans1d.cluster(_freq_arr_val, len(alphabet))

    # build replacement map
    _map = {}
    for i in range(len(_freq_arr_key)):
        _map[_freq_arr_key[i]] = clusters[i]

    # Transform ciphertext to substitution cipher
    _out = ""
    for p in ciphertext:
        _out += alphabet[_map[key(p)]]

    return _out
```

Advanced Attack: 1 plaintext - n ciphertext

The encryption is not injecting a nonce; therefore, it always encrypts a unique plaintext. This attack scheme aims at analyzing different ciphertext based on the same plaintext to remove the random distribution of multiple pixels representing a single character. Especially on large plain/ciphertexts, this allows an easy grouping of pixels unlinked to their occurrence. A common pattern is, e.g., an e-mail header.

One approach for grouping pixels has been implemented to validate the attack. In a first iteration, multiple ciphertexts are an analyzed character after character. In a first step, edges of a graph are extracted (`pixel_1 -> pixel_2`). In a second step, edges are followed and joined in extracting disjoint sets of graph vertices (in this case: pixels).

Finally, the ciphertext is reduced to a simple substitution cipher, which can be cracked, e.g., using trivial frequency analysis. This algorithm is not fast but acts as a proof of concept of this attack.

```python
def substitute(ciphertexts, alphabet=ALPHABET, analysis_size=10000):
```

```
_edges = set()

for i in range(len(ciphertexts[0][:analysis_size])):
    # add an edge to the list

    # Extract pixel sharing this group
    p = key(ciphertexts[0][i])

    for _i in range(1, len(ciphertexts)):
        k = key(ciphertexts[_i][i])
        _edges.add(
                (p, k) if p < k else (k, p)
        )

groups = []
while len(_edges) > 0:
    initial = _edges.pop()
    _group = set(list(initial))

    print(f"[+] len(edges)={len(_edges)}")

    added = True
    while added:
        # For every group find all sub-edges
        added = False
        _toadd = set()
        _toremove = set()
        for p in _group:
            for e in _edges:
                if p in list(e):
                    for i in list(e):
                        _toadd.add(i)
                    _toremove.add(e)
                    added = True
        print("[+] merging...")
        for i in _toadd:
            _group.add(i)
        for i in set(_toremove):
            _edges.remove(i)
        print(f"[+] merged {len(_toremove)} edges, {len(_edges)} remaining")


    print(f"[+] successfully extracted group with size {len(_group)}")
    groups.append(_group)
    print(f"[+] total number of groups: {len(groups)} ")
print(f"Done; len(groups)={len(groups)}")

# Generate translate map
print(f"Generate translate map {len(groups)} -> {len(ALPHABET)}")
_map = {}
for i in range(max(len(ALPHABET),len(groups))):
    for p in groups[i]:
        _map[p] = ALPHABET[i]

# Substitute ciphertext to substitution cipher.
_out = ""
for p in ciphertexts[0]:
    _out += _map[p]

print(f"Substituted ciphertext with {len(_out)} characters")
return _out
```

Advanced Attack: Adding known-plaintext attack

The attack, as mentioned above, allows us to group pixels based on many ciphertexts. While a simple frequency analysis can effectively solve the cipher, a known plaintext-ciphertext pair helps to crack a plaintext that is not vulnerable to frequency analysis.

Frequency Analysis

Based on the extracted pixel groups, a rather simple frequency analysis extracts the key. First, the letter frequency is computed; afterward, we map it to the desired frequency.

```
def frequency_map(ciphertext):
    """ Extracts letter frequency """
    total = len(ciphertext)

    # Compute occurence
    _out = {}
    for c in ciphertext:
        _out[c] = _out.get(c, 0) + 1

    # Compute percentage
    out={}
    for k, v in _out.items():
        out[k] = float(v) / float(total)

    return sorted(out.items(), key=operator.itemgetter(1), reverse=True)
```

```python
def get_key(ciphertext):
    """ Simple mapping of letter frequency to desired frequency"""

    freq = "".join([c for c, _ in frequency_map(ciphertext)])
    key = str.maketrans(freq, DESIRED_FREQ)

    return key
```

Extracting the key image

It is impossible to extract the original image as the pixel value space is reduced from 8 to 7 bits. However, mapping the cracked key to an image may help identify the original image's structures while generating a valid key.