Quebrando 'Two-Time Pads' com Ataque de Dicionário e *Crib Drag*

Universidade Estadual de Campinas

Felipe Santos Oliveira
RA 119383

7 de abril de 2015

1 Introdução

One-time pad (OTP) são stream-ciphers onde dado uma mensagem m de tamanho n e uma chave k, onde $|k| \geq n$ e é aleatoriamente gerada, $m_i \oplus k_i = c_i$, onde c_i é o i-ésimo bit da mensagem m cifrada com a chave k pela operação de *ou exclusivo* ($\oplus$ ou XOR). O OTP possui a propriedade de sigilo perfeito de Shannon (*perfect secrecy*), ou seja, para cada k_i existem $|k|$ possibilidades para que $k_i \oplus c_i = m_i$. Essa propriedade só será válida para o OTP se e somente se a chave k for usada para uma única mensagem. Nesse experimento realizaremos a criptanálise de uma variante do OTP, o two-times pad (TTP), onde basicamente se usa uma mesma chave k para cifrar duas mensagens, m_1 e m_2, distintas.

2 Metodologia

Sejam m_1 e m_2 as mensagem em texto puro, k a chave e $c1$ e $c2$ as cifras de $m1$ e $m2$ respectivamente, tais que:

$m_1^i \oplus k = c_1^i$
$m_2^i \oplus k = c_2^i$

No entanto, quando usamos a mesma chava para as duas mensagens, ao fazer XOR entre as duas cifras, teremos:

$$c_1^i \oplus c_2^i = (m_1^i \oplus k) \oplus (m_2^i \oplus k) = m_1^i \oplus m_2^i$$

Portanto, ao usar a mesma chave para as duas mensages, criamos a vulnerabilidade de que fazer XOR entre as duas mensagens cifradas nos dá o mesmo resultado de fazer XOR com as duas mensagens em texto puro. Isso abre espaço para a realização de *cribbing* ou *crib drag*, que consiste em chutar uma palavra com alta probabilidade de estar no texto e realizar o XOR dessa palavra com o resultado do XOR das duas cifras. Caso a palavra esteja em uma das mensagens originais, essa operação deve retornar uma palavra ou pedaço de palavra presente naquela posição na outra mensagem. Fazendo isso sucessivamente, podemos extrair pedaços cada vez maiores das mensagens até que obtermos o conteúdo da menor das duas mensagens, caso sejam de tamanhos diferentes, ou as mensagens originais completas caso sejam de tamanho igual.

No experimento realizado, as mensagens possuiam 202 e 204 caracteres respectivamente. Portanto, só se é possível obter 202 caracteres da mensagem maior usando métodos automatizados, a não ser que se faça um *padding* na menor.

Inicialmente tentou-se uma abordagem de rasterização, onde para cada palavra do dicionário se testariam todas as posicões do texto até se achar um match. A partir daí tentaria-se expandir o resultado, indo para a próxima palavra do dicionário no caso de se esbarrar em deadend. Isso se mostrou inefetivo do ponto de vista automático, dado que o problema de decisão de em que mensagem o resultado deve ir era mais complexo que o esperado.

Para realizar o *crib drag* de maneira automatizada, então, foi criado um algoritimo recursivo em que se passa inicialmente duas strings vazias, o resultado do XOR das duas cifras, um cursor, uma string vazia para armazenar o resultado encontrado na última chamada da função e o tamanho da menor cifra. A condição de para é encontrar uma mensagem do mesmo tamanho da menor cifra que seja uma frase bem construída.

A cada chamada, usamos o que encontramos até agora no primeiro resultado para chutar a próxima palavra da frase. Se o resultado do XOR da palavra com o XOR das cifras for uma palavra, um prefixo ou um conjunto de palavras válidas com ou sem um prefixo no final, chamamos a função recursiva passando dessa vez o segundo texto contatenado com o resultado como primeiro argumento e o primeiro texto concatenado com a palavra sendo chutada. Fazemos isso até a condição de parada ser alcançada ou percorrermos todo o dicionário para um dado chute. Caso estouremos o dicionário, realizamos um *backtrack* e tentamos a próxima palavra em uma posição anterior. Caso não achemos nada, retornamos strings vazias.

O algorítimo em pseudo-código se encontra a seguir:

Algorithm 1 Find Messages

```
procedure FindMessages(ptext1, ptext2, cipherxor, cursor, previousCursor, currentPart, dictionary)
    if ptext1.lenght == smallerCipher.lenght or ptext2.lenght == smallerCipher.lenght: then
        if one of the plaintextes is a well formed sentece: then
            return (ptext1, ptext2)
    else
        for word in dictionary do
            result = (ptext1 + word + ' ') ⊕ cipher
            if result is not empty and there is a word starting with result then
                answer = FindMessage(ptext2 + result, ptext1 + word
                                     cipherxor, cursor + word.lenght, cursor, result, dictionary)
                if answer.ptext1 != ptext1 then
                    return answer
```

3 Implementação

A linguagem escolhida para a implementação do algorítimo projetado foi Python devido a suas facilidades em manipulação de strings. Deve-se ressaltar que a implementação não visou ser tempo-eficiente, portanto pode ser lenta em casos mais extremos que o caso de estudo. A implementação encontra-se entre as linhas 108 e 157 do arquivo *stream_ crib.py*.

Para a automação do *cribbing* foi utilizado um dicionário das 10.000 palavras mais frequentes do inglês, carregado em duas estruturas diferentes: um *map* para checar se uma dada string é uma palavra no meu dicionário, e uma lista usada para as iterações em ordem de frequência na recursão.

Como foi dito antes, a implementação não visa velocidade, e sim eficácia na tarefa proposta, mas estruturas como *trie* ou *radix tree* poderiam ser usadas em vez de uma lista afim de acelerar certas partes das verificações.

Para essas, foram implementadas três funções auxiliares: *testCrib*, que retorna o resultado do XOR entre uma string usada como crib e outra como alvo e será do tamanho do crib; *hasWordStartingWith*, que verifica se em dada string há um prefixo válido para palavras no dicionário e retorna um booleano e *WordStartingWith*, que busca uma palavra começada com um dado prefixo.

Devido ao fato de a mensagem em texto puro do caso de estudo ser uma frase mal-formada, acabando em um espaco e uma quebra de linha, achar uma condição de parada foi um pouco problemático. A condição usada é a descrita no algorítimo, onde uso o tamanho da menor mensagem como verificação do termino, assim como os caractéres finais.

4 Resultados

Como uma das mensagens é 2 caractéres maior que a outra, esses não podem ser recuperados automaticamente, a não ser por extrapolação da palavra final. O algorítimo funciona consideravelmente bem, encontrando as palavras corretas e se autocorrigindo em caso de erro.

```
-----
1: 'I can't in good conscience allow the U.S. government to destroy privacy, internet
freedom and basic liberties for people around tm a'
2: 'Taken in its entirety, the Snowden archive led to an ultimately simple conclusion:
the U.S. government had built a system that have '
Cursor: 132
-----
```

```
-----
1: 'I can't in good conscience allow the U.S. government to destroy privacy, internet
freedom and basic liberties for people around zv a'
2: 'Taken in its entirety, the Snowden archive led to an ultimately simple conclusion:
the U.S. government had built a system that home '
Cursor: 132
-----
```

```
-----
1: 'I can't in good conscience allow the U.S. government to destroy privacy, internet
freedom and basic liberties for people around the'
2: 'Taken in its entirety, the Snowden archive led to an ultimately simple conclusion:
the U.S. government had built a system that has '
Cursor: 131
-----
```

Acima pode se ver um trecho da execução do programa com verbose ativa onde ocorre uma autocorreção do texto. Ao final, recuperamos 202 caracteres dos 205 caracteres da chave e as duas mensagens:

```
Text 1: 'I can't in good conscience allow the U.S. government to destroy privacy,
internet freedom and basic liberties for people around the world with this massive
surveillance machine they're secretly building. '

Text 2: 'Taken in its entirety, the Snowden archive led to an ultimately simple
conclusion: the U.S. government had built a system that has as its goal the complete
elimination of electronic privacy worldwide.
'
```

Mais dois caracteres da chave podem ser obtidos uma vez que se tem a mensagem de 204 caractéres, totalizando 204 de 205 caractéres da chave obtidos.

Para efeito de comparação, ao executar o código usando o comando *time* do UNIX, foram obtidos os seguintes tempos:

```
real 0m0.343s
user 0m0.330s
sys 0m0.016s
```

5 Conclusão

Durante o estudo, foram utilizadas diversas abordagens para a automação do ataque ao TTP, mas a que se mostrou mais efetiva foi a implementada, dado que não foi necessária implementação de um algorítimo de decisão para saber onde cada palavra deve ir. Finalmente, a obtenção tanto dos textos quanto da chave usada na cifragem mostra a vulnerabilidade do TTP.