Szyfrowanie transmisji sieciowej przy użyciu chaosu

Andrzej Chrzan
Paweł Gumola
Kamil Woźniak
Jakub Woźny

8 II 2016

Spis treści

1 Wstęp

Zadanie polegało na napisaniu programu w Pythonie szyfrującego i deszyfrującego wiadomości podczas transmisji przez sieć przy użyciu metody reverse interval mapping oraz varaible bit length encoding.

2 Teoria

2.1 Historia

Pierwsze odkrycia dotyczące chaosu można przypisać Hadamardowi, który opublikował w 1898 roku pracę dotyczącą bil poruszających się bez tarcia po powierzchni o ujemnej krzywiźnie. Hadamard pokazał, że w takich warunkach wszystkie trajektorie są niestabilne w tym sensie, że oddalają się od siebie wykładniczo, z dodatnim wykładnikiem Lapunowa. Na początku XX wieku Henri Poincare pokazał, że w problemie n-ciał istnieją orbity, które są aperiodyczne, ale nie są zbieżne ani rozbieżne. Problem ten był badany w kolejnych latach przez wielu matematyków i fizyków. Efektem tych prac było pokazanie podobnego zachowania dla wielu układów, takich jak turbulentne przepływy i oscylacje w obwodach elektrycznych. Zbudowanie teorii opisującej te zjawiska wymagało jednak dopiero zastosowania symulacji komputerowych. Pionierem teorii chaosu stał się Edward Lorenz, który w 1961 przeprowadzał numeryczne analizy zjawisk pogodowych. Symulowany przez niego układ opisywał własności ogrzewanej, prostokątnej komórki gazowej. Składał się z pięciu równań różniczkowych nieliniowych, będących ograniczoną wersją równań Naviera-Stokesa. Lorenz, chcąc uprościć obliczenia przerwane błędem sprzętowym, zamiast przeprowadzać je od początku, rozpoczął kontynuację symulacji od wyników pośrednich uzyskanych przed momentem awarii. Jak zauważył pod koniec, otrzymane wyniki w znaczny sposób odbiegały od symulacji przeprowadzonych od początku do końca. Okazało się to skutkiem zaokrąglenia wprowadzanych ręcznie wyników. Równania okazały się zaskakująco czułe na niewielką zmianę warunków początkowych.

2.2 Atraktory

Niektóre układy dynamiczne są chaotyczne wszędzie, ale w większości wypadków takie zachowanie dotyczy jedynie pewnego podzbioru przestrzeni fazowej. Najbardziej interesujący przypadek zachodzi, gdy chaotyczność dotyczy jakiegoś atraktora, gdyż trajektorie z całego jego obszaru przyciągania mają tę własność. Atraktory w układach liniowych są zwykle punktami lub okręgami. W układach chaotycznych pojawiają się dziwne atraktory o bardzo złożonej budowie, często fraktalnej. Jednym z najsłynniejszych przykładów jest trójwymiarowy atraktor Lorenza, przypominający kształtem motyla. W celu badania własności chaosu rozwinięto wiele technik w zakresie analizy równań różniczkowych oraz wykorzystano w nowy sposób wiele istniejących narzędzi matematycznych. Na potrzeby symulacji komputerowych dla układów chaotycznych korzysta się z przekrojów Poincarego, umożliwiających zmniejszenie wymiaru przestrzeni fazowej. Następnie z własności tych przekrojów wnioskuje się na temat własności pełnej przestrzeni fazowej rozwiązań.

2.3 Chaos deterministyczny

Chaos deterministyczny to własność równań lub układów równań, polegająca na dużej wrażliwości rozwiązań na dowolnie małe zaburzenie parametrów. Dotyczy to zwykle nieliniowych równań różniczkowych i różnicowych, opisujących układy dynamiczne. Przesłankę prowadzącą do sformułowania teorii chaosu były badania Edwarda Lorenza nad modelami prognozowania pogody. Zgodnie z ówczesnym, deterministycznym rozumieniem rzeczywistości minimalna zmiana warunków początkowych powinna prowadzić do proporcjonalnie niewielkich zmian wyniku modelu. W trakcie pracy nad modelem, z natury dynamicznym (dane z iteracji wcześniejszych są danymi wejściowymi dla iteracji następujących), w celu ułatwienia pracy wprowadził zaokrąglone wartości wyjściowe. Okazało się, że wynik modelu diametralnie odbiegał od tego co przewidywał ten sam model przy danych wprowadzonych z większą dokładnością. Dalsze badania nad układami dynamicznymi doprowadziły do wniosku, iż wbrew powszechnym przekonaniom w nauce, niewielkie zaburzenie warunków początkowych powoduje rosnące wykładniczo z czasem zmiany w zachowaniu układu. Popularnie nazywane jest to efektem motyla - znikoma różnica na jakimś etapie może po dłuższym czasie urosnąć do dowolnie dużych rozmiarów. Powoduje to, że choć model jest deterministyczny, w dłuższej skali czasowej wydaje się zachowywać w sposób losowy. Zachowanie takie możżna zaobserwować w wielu zjawiskach fizycznych, między innymi w zmianach pogody, oscylujących reakcjach chemicznych, zachowaniu niektórych obwodów elektrycznych i ruchu ciał oddziałujących grawitacyjnie.

2.4 Mapy jednowymiarowe

Rozpatrujemy iterację jednowymiarowych map: $f : [0,1] \to [0,1]$ w postaci $x_{t+1} = rf(x_t), t = 0,1,2...$

gdzie
$r > 1$;
$f \in [0,1]$
monotoniczne na [0,0.5]

Przykłady takich map to:

- mapa Bella
- mapa entropii
- mapa logistyczna
- mapa namiotu
- mapa sinusoidalna

2.5 Chaotyczna transmisja danych

Symetryczna mapa namiotów f :[-1, +1] -> [-1, +1]

$$f(x) = \begin{cases} \text{2x+1} \\ \text{gdy } -1 \leq x \leq 0 \\ \text{-2x+1} \\ \text{gdy } 0 \leq x \leq 1 \end{cases}$$

Jest to w zupełności chaotyczna mapa z niezmienną gęstością ρ = 1/2 i wykładnikiem Ljapunova λ = ln(2). Mamy zatem iterację $x_{t+1} = f(x_t)$, gdzie t = 0,1, ... i $x_0 \epsilon [-1,+1]$, jest wartością początkową

Generujemy ciąg o długości T z mapy, np. $x_0, x_1, ..., x_{T-1}$. Będzie to nadajnik. Mając teraz łańcuch bitów $b = (b_0, b_1, ..., b_{T-1})$ dla sygnału o długości T, gdzie $b_t[-1,+1]$, formujemy transmitowany sygnał $s = (s_0, s_1, ..., s_{T-1})$

$s_t = b_t x_t$, t=0,1,.....,T-1

Odbiornik jest teraz dany jako
$y_{t+1} = f(s_t)$, t=0,1,.....,T-2

Oryginalną sekwencję bitów b może odnaleźć formując iloczyny

$s_t y_t$, t=0,1,.....,T-1

Jeżli $s_t y_t > 0$ to $b_t = 1$ i jeżli $s_t y_t < 0$ to $b_t = -1$. O to dowód
$s_t y_t = s_t f(s_{t-1})$
$= s_t f(b_{t-1} x_{t-1})$
$= s_t f(x_{t-1}) sine f(x_t) = f(-x_t)$
$= b_t x_t f(x_{t-1})$
$= b_t x_t^2$
W konsekwencji
$sign(s_t y_t) = sign(b_t x_t^2) = b_t$

Schemat ten nie narzuca limitu długości ciągu bitów, ponieważ rozbieżność ciągu daje tylko lokalne błędy. Wartości s_t sekwencji są transmitowane i bezpośrednio operuje na nich odbiorca.

2.6 Reverse interval mapping - podstawy

Wiadomości kodowane w formie:

$m = m_1 m_2 ... m_n \epsilon \Sigma_2^*$

Wartości na przedziale [0,1] są związane z Σ_2 z mapą d : 0,1 -> Σ_2

d(x) = $\begin{cases} 0 & \text{gdy } 0 \leq x \leq \frac{1}{2} \\ 1 & \text{gdy } \frac{1}{2} < x \leq 1 \end{cases}$

Na początku zaczynamy z dowolną wartością x_0 spoza przedziału [0,1]. Przykładowo $\frac{(1+r)}{2}$ jest wygodnym wyborem. Wartość ta oznacza początek wiadomości i koniec procesu dekodowania. Punkt ten posiada dwa odwrócone obrazypod mapą rf.
$x_{1,0} = f^{-1}(\frac{x_0}{r}), x_{1,1} = 1 - f^{-1}(\frac{x_0}{r})$

Postępujemy w ten sposób aż do osiągnięcia maksymalnejprecyzji rejestru przechowującego wartość x_i albo do momentu, kiedy wszystkie wiadomości zostały zakodowane

$x_0 := \frac{1+r}{2}$

$$x_1 = \begin{cases} \text{f}^{-1}(\frac{x_0}{r}) \text{ gdy } m_1 = 0 \\ \text{1-f}_{-1}(\frac{x_0}{r}) \text{ gdy } m_1 = 1 \end{cases}$$

$$x_2 = \begin{cases} \text{f}^{-1}(\frac{x_1}{r}) \text{ gdy } m_2 = 0 \\ \text{1-f}^{-1}(\frac{x_1}{r}) \text{ gdy } m_2 = 1 \end{cases}$$

2.7 Variable bit length encoding - podstawy

W metodzie tej zamiast iteracji pojedynczego numeru iteruje się cały przedział.

$[a, b] \subset (f^{-1}(\frac{1}{r}), 0.5)$ lub $[a, b] \subset (0.5, 1 - f^{-1}(\frac{1}{r}))$

Tak więc zaczynamy z przedziaęem [a, b], który wyiteruje się z [0,1] pod r f. Wygodnym wyborem w tym wypadku jest

$w := 0.1, x_0 := f^{-1}(\frac{1}{r}), a = (1 - w)x_0 + \frac{w}{2}, b = wx_0 + \frac{1-w}{2}$

gdzie 0 < w < 0.5 jest wagą do średniej ważonej $x_0 = f^{-1}(\frac{1}{r})$ i 0.5, do obliczenia a i b. Mniejsze w jest preferowane, ponieważ daje większy przedział początkowy, który następnie się zmniejszy.

Metoda znalezienia kolejnych przedziałów jako:
$a_0 = (1 - w)x_0 + \frac{w}{2}$

$$a_1 = \begin{cases} \text{f}^{-1}(\frac{a_0}{r}) \text{gdy } m_1 = 0 \\ \text{1-f}_{-1}(\frac{b_0}{r}) \text{gdy } m_1 = 1 \end{cases}$$

$$a_2 = \begin{cases} \text{f}^{-1}(\frac{a_1}{r}) \text{gdy } m_2 = 0 \\ \text{1-f}^{-1}(\frac{b_1}{r}) \text{gdy } m_2 = 1 \end{cases}$$

$b_0 = wx_0 + \frac{1-w}{2}$

$$b_1 = \begin{cases} \text{f}^{-1}(\frac{b_0}{r}) \text{gdy } m_1 = 0 \\ \text{1-f}_{-1}(\frac{a_0}{r}) \text{gdy } m_1 = 1 \end{cases}$$

$$b_2 = \begin{cases} \text{f}^{-1}(\frac{b_1}{r}) \text{gdy } m_2 = 0 \\ \text{1-f}^{-1}(\frac{a_1}{r}) \text{gdy } m_2 = 1 \end{cases}$$

Kontynuowane jest dopóki $b_n - a_n < \epsilon$. Warto dostrzec że $m_j = 1$ powoduje zmiane w rolach a_j i b_j tak, że wartość bezwzględna jest zawsze dana przez $b_j - a_j$ i tak, że otrzymany przedział leży [0.5, 1].

Kiedy warunek $b_n - a_n < \epsilon$jest spełniony, przechowujemy a_n na końcu sekwencji liczb reprezentujących dane i zaczynamy jeszcze raz przezzdefiniowanie na nowo a_n i b_n

$a_n = (1 - w)x_0 + \frac{w}{2}$

$$a_{n+1} = \begin{cases} \text{f}^{-1}(\frac{a_n}{r})\text{gdy } m_{n+1} = 0 \\ \text{1-f}_{-1}(\frac{b_n}{r})\text{gdy } m_{n+1} = 1 \end{cases}$$

$$a_{n+2} = \begin{cases} \text{f}^{-1}(\frac{a_{n+1}}{r})\text{gdy } m_{n+2} = 0 \\ \text{1-f}^{-1}(\frac{b_{n+1}}{r})\text{gdy } m_{n+2} = 1 \end{cases}$$

$b_n = wx_0 + \frac{1-w}{2}$

$$b_{n+1} = \begin{cases} \text{f}^{-1}(\frac{b_n}{r})\text{gdy } m_{n+1} = 0 \\ \text{1-f}_{-1}(\frac{a_n}{r})\text{gdy } m_{n+1} = 1 \end{cases}$$

$$b_{n+2} = \begin{cases} \text{f}^{-1}(\frac{b_{n+1}}{r})\text{gdy } m_{n+2} = 0 \\ \text{1-f}^{-1}(\frac{a_{n+1}}{r})\text{gdy } m_{n+2} = 1 \end{cases}$$

Dopóki $b_n + k - a_n + k < \epsilon$., a potem kontynuujemy proces aż wszystkie wartości m_j zostały użyte. Zatem kodowanie daje teraz wiele liczb rzeczywistych h, z których każda reprezentuje kawałek ciągu wiadomości , gdzie każda z liczb rzeczywistych niekoniecznie musi kodować taką samą liczbę symboli.

2.8 Działanie programu

Aby działał program najpierw musimy stworzyć wirtualny serwer. W tym celu w konsoli wpisujemy komendę python server.py. Po wpisaniu tej komendy serwer sie załącza i poprosi nas o sprecyzowanie jak będziemy chcieli kodować pliki. Wyświetla nam się informacja na temat dwóch typów kodowania: vbl i rim, a także adresu portu serwera. Przykładowo wybieramy kodowanie rim. Następnym krokiem jest załączenie pierwszego użytkownika z naszym serwerem. Do tego otwieramy kolejną konsolę i tym razem wpisujemy komendę python client.py 9009 <usr>. Na konsoli pokazuje nam się informacja, że zostaliśmy połączeni z serwerem. Serwer odhaczył, że uytkownik się z nim połączył. Do komunikacji potrzebujemy jeszcze kolejnego użytkownika. Więc postępujemy analogicznie jak przy pierwszym. Gdy są już co najmniej dwie osoby połączone z serwerem można rozpocząć przesyłanie wiadomości między nimi. Jeden z użytkowników pisze wiadomość do drugiego. Zanim druga osoba otrzyma jego wiadomość przechodzi przez serwer który koduje wiadomość w system binarny (1 znak to 8 bitów),

a następnie przekazuje zakodowaną wiadomość do drugiego użytkownika. Oczywiście drugi użytkownik może też komunikować się z pierwszym, na takiej samej zasadzie. Drugie kodowanie to VBL, które działa a bardzo w analogiczny sposób, kodowanie na serwerze odbywa siwę w mniejszej ilości bitów.
Dodatkowe informacje:
Po wylogowaniu się użytkownika, drugi użytkownik dalej zostaje połączony z serwerem. Po zamknięciu serwera wszyscy użytkownicy zostają automatycznie rozłączeni.

3 Kod

3.1 Reverse interval mapping

Kodowanie

Objaśnienie:
4 znaki są równoważne 32 bitom.
Wartość parametru "r" jest stała podczas kodowania i odkodowania.
Parametr "msg" jest konwertowany na bitowy cią.
Parametr "return" jest to wartość wyliczona "msg".

```
def reverse_interval_mapping_encode(msg, r=1.025):
x = (1.0 + r) / 2
for m in msg:
if m == ’0’:
x = inverse_logistic_map(x / r)
else:
x = 1.0 - inverse_logistic_map(x / r)
return x
```

Odkodowanie

Objaśnienie:
Parametr "x" jest to zakodowana wiadomość

```
def reverse_interval_mapping_decode(x, r=1.025):
m = ”
while 0.0 <= x <= 1.0:
if x < 0.5:
m = ’0’ + m
else:
m = ’1’ + m
x = r * logistic_map(x)
return m
```

```
    def rim_encode(text):
binary_text = to_binary(text)
splitted = [binary_text[i:i + 3] for i in xrange(0, len(binary_text), 3)]
text_doubles = []
```

```
for double in splitted:
text_doubles.append(reverse_interval_mapping_encode(double))
return ' '.join(map(str, text_doubles))

def rim_decode(encoded_text):
text_doubles = map(float, encoded_text.split())
binary_text = ''
for text_double in text_doubles:
binary_text += reverse_interval_mapping_decode(text_double)
return to_text(binary_text)
```

3.2 Variable bit length

Kodowanie

```
def variable_bit_length_encode(msg, r=1.025, epsilon=0.0000000001):

a = 0.0
b = 0.0
w = 0.1
x_0 = inverse_logistic_map(1.0 / r)
doubles = []

for m in msg:
if b - a < epsilon:
a = (1.0 - w) * x_0 + w * 0.5
b = w * x_0 + (1.0 - w) * 0.5
doubles.append(a)

if m == '0':
doubles[len(doubles) - 1] = a
else:
doubles[len(doubles) - 1] = 1.0 - a
a = 1.0 - b
b = doubles[len(doubles) - 1]

a = inverse_logistic_map(a / r)
b = inverse_logistic_map(b / r)

return doubles
```

Odkodowanie

```
def variable_bit_length_decode(doubles, r=1.025):
message = ”
while len(doubles) != 0:
double = doubles.pop()
while 0.0 <= double <= 1.0:
if double < 0.5:
message = ’0’ + message
else:
message = ’1’ + message
double = r * logistic_map(double)
return message
```

```
    def vbl_encode(text):
binary_text = to_binary(text)
text_doubles = variable_bit_length_encode(binary_text)
return ’ ’.join(map(str, text_doubles))
```

```
    def vbl_decode(encoded_text):
text_doubles = map(float, encoded_text.split())
binary_text = variable_bit_length_decode(text_doubles)
return to_text(binary_text)
```

4 Podsumowanie

W danym projekcie stworzyliśmy aplikacje, która zapewnia komunikacje pomiędzy dwoma użytkownikami. Użyta został metoda zabezpieczenia transmisji danych przed ich ujawnieniem. Wdrożenie mechanizmów ochrony kryptograficznej gwarantuje integralność przesyłanych informacji oraz potwierdza, że druga strona komunikacji jest tą, za którą się podaje.

Literatura

[1] Willi-Hans Steeb; The Nonlinear Workbook

[2] S. Sternberg; Dynamical systems `http://www.math.harvard.edu/library/sternberg/`

[3] `http://www.bogotobogo.com/python/python_network_programming_tcp_server_client_chat_server_chat_client_select.php`