Dokumentacja końcowa projektu
Algorytm DSA

Jakub Turek

Podstawy teoretyczne

FIPS[1] jest zbiorem, w którym opisane są publiczne standardy bezpieczeństwa używane przez federalny rząd Stanów Zjednoczonych. Oficjalnym standardem podpisywania wiadomości cyfrowych zamieszczonym w FIPS jest DSS (ang. Digital Signature Standard). DSS opiera się o algorytm DSA (ang. Digital Signature Algorithm).

Standard DSS (wraz z algorytmem DSA) został opisany w dokumencie FIPS PUB 186[2]. Na potrzeby projektu zaimplementowany został oryginalny algorytm opublikowany w 1994 roku, który wykorzystuje funkcję skrótu SHA.

Algorytm

Generacja kluczy Należy wybrać parametry:

- Liczba pierwsza p w pierścieniu reszt modulo a, gdzie $2^{L-1} < p < 2^L$ oraz $512 \leqslant L \leqslant 1024$ i L jest wielokrotnością 64.
- Liczba pierwsza q będąca dzielnikiem liczby $p-1$ w pierścieniu reszt modulo a, gdzie $2^{159} < q < 2^{160}$.
- Liczba $g = h^{\frac{p-1}{q}} \pmod p$, gdzie h jest dowolną liczbą naturalną spełniającą warunek $1 < h < p-1$ taką, że $h^{\frac{p-1}{q}} \pmod p > 1$ (czyli g ma rząd $q \pmod p$).
- Losowo wygenerowana liczba x z przedziału $0 < x < q$.
- Liczba $y = g^x \pmod p$.
- Losowo wygenerowana liczba k z przedziału $0 < k < q$.

Liczby p, q oraz g są publiczne. Klucz prywatny użytkownika to x, natomiast klucz publiczny użytkownika to y. Parametr k musi być obliczany dla każdego nowego podpisu. Klucze są wielokrotnego użytku.

Podpisywanie wiadomości Podpisem wiadomości M jest para liczb (r, s) obliczanych według poniższego wzoru:

$$\begin{aligned} r &= (g^k \mod p) \mod q \\ s &= (k^{-1}(SHA(M) + xr)) \mod q \end{aligned}$$

gdzie k^{-1} jest odwrotnością liczby k w pierścieniu reszt modulo q (czyt. $k \cdot k^{-1} \equiv 1 \pmod q$).

Opcjonalnie można zweryfikować, czy $r \neq 0$ i $s \neq 0$. Jeżeli jeden z warunków nie jest spełniony, należy wygenerować podpis od nowa. Sytuacja taka nie powinna się jednak zdarzyć dla prawidłowo wygenerowanych kluczy.

[1] Skrót od **F**ederal **I**nformation **P**rocessing **S**tandard (ang. federalny standard przetwarzania informacji).
[2] `http://www.itl.nist.gov/fipspubs/fip186.htm`.

Weryfikacja podpisu Zakładamy, że otrzymaliśmy zestaw $(M', (r', s'))$ składający się z wiadomości oraz podpisu tej wiadomości. Aby zweryfikować podpis należy:

1. Dokonać sprawdzenia, że $0 < r' < q$, a ponadto $0 < s' < q$.
2. Obliczyć poniższe wartości:

$$\begin{aligned} w &= (s')^{-1} \pmod q \\ u_1 &= SHA(M') \cdot w \pmod q \\ u_2 &= r' \cdot w \pmod q \\ v &= (g^{u_1} \cdot y^{u_2} \mod p) \pmod q \end{aligned}$$

3. Sprawdzić, czy $v = r'$.
4. Podpis jest prawidłowy, jeżeli warunek z punktu 3. jest spełniony.

Dowód poprawności

Lemat Niech p i q będą liczbami pierwszymi takimi, że q dzieli $p-1$, h jest dodatnią liczbą całkowitą mniejszą niż p i spełnione jest równanie $g = h^{\frac{p-1}{q}} \mod p$. Wtedy $g^q \mod p = 1$ i, jeżeli $m \mod q = n \mod q$, wtedy $g^m \mod p = g^n \mod p$.

Dowód Z Małego Twierdzenia Fermata wynika równość:

$$g^q \mod p = (h^{\frac{p-1}{q}} \mod p)q \mod p = h^{p-1} \mod p = 1$$

Jeśli $m \mod q = n \mod q$, przykładowo $m = n + kq$ dla pewnej liczby całkowitej k. Wtedy:

$$g^m \mod p = g^{n+kq} \mod p = (g^n g^{kq}) \mod p = ((g^n \mod p)(g^q \mod p)^k) \mod p = g^n \mod p$$

ponieważ $g^q \mod p = 1$.

Twierdzenie Jeżeli $M' = M$, $r' = r$ oraz $s' = s$ w podpisie DSA, wtedy $v = r'$.

Dowód Mamy:

$$\begin{aligned} w = &\quad (s')^{-1} \mod q &&= s^{-1} \mod q \\ u_1 = &\quad (SHA(M') \cdot w) \mod q &&= (SHA(M) \cdot w) \mod q \\ u_2 = &\quad (r' \cdot w) \mod q &&= r \cdot w \mod q \end{aligned}$$

$y = g^x \mod p$, zatem wykorzystując lemat:

$$\begin{aligned} v &= (g^{u_1} y^{u_2} \mod p) \mod q = \\ &= (g^{SHA(M)\cdot w} y^{r\cdot w} \mod p) \mod q = \\ &= (g^{SHA(M)\cdot w} g^{xrw} \mod p) \mod q = \\ &= (g^{(SHA(M)+xr)\cdot w} \mod p) \mod q \end{aligned}$$

Ponadto wiemy, że $s = (k^{-1}(SHA(M) + xr)) \mod q$. Stąd wynika, że:

$$\begin{aligned} w = (k(SHA(M) + xr)^{-1}) \mod q \\ ((SHA(M) + xr) \cdot w) \mod q = k \mod q \end{aligned}$$

Zatem, powołując się na lemat, możemy stwierdzić, że:

$$v = (g^k \mod p) \mod q = r = r'$$

co należało udowodnić.

Implementacja

Implementacja została oparta o sugestie zawarte w załącznikach do dokumentu FIPS PUB 186. Przedstawiony został w nich algorytm generacji kluczy. Dodatkowo, do implementacji wykorzystane zostały komponenty biblioteki `PyCrypto`[3]. Dzięki temu możliwe było uniknięcie implementacji funkcjonalności, które są przedmiotami innych projektów, takich jak funkcja skrótu `SHA` czy test pierwszości Millera-Rabina.

Generacja liczb p i q Załącznik drugi (punkt 2.2) do dokumentu FIPS PUB 186 sugeruje, aby w pierwszej kolejności wygenerować liczbę q, w oparciu o dowolną sekwencję długości przynajmniej 160 bitów nazwaną ziarnem.

`seed = os.urandom(20)`	Wybieramy losowy ciąg znaków o długości 20 bajtów (160 bitów).

`hash1 = SHA.new(seed).digest()`
`hash2 = SHA.new(long_to_bytes(`
`  bytes_to_long(seed)+1)).digest()`
	Obliczamy funkcję skrótu dla ziarna...
	... oraz ziarna powiększonego o 1.

`q = 0L`	
`for i in range(0, 20):` `  c = ord(hash1[i]) ^ ord(hash2[i])`	Dla każdego bajtu ziarna... ... obliczamy wartość funkcji XOR.
`  if i == 0:` `  c |= 128`	Jeżeli najbardziej znaczący bajt... ... to ustawiamy jeden na pierwszym bicie.
`  if i == 19:` `  c |= 1`	Jeżeli najmniej znaczący bajt... ... to ustawiamy jeden na ostatnim bicie.
`  q = q * 256 + c`	Przesuwamy obliczoną liczbę o bajt w lewo i dołączamy z prawej obliczony w pętli bajt.
`while not isPrime(q):` `  q += 2`	Dodajemy kolejne dwójki (nie zmienia się ostatni bit) aż do uzyskania liczby pierwszej.
`if pow(2, 159L) < q < pow(2, 160L):` `  return seed, q`	Jeżeli uzyskane q zawiera się w narzuconym przedziale to zwracamy ziarno oraz q.

W stosunku do części algorytmu generacji q przedstawionej w krokach 1. - 5. zastosowano jedną optymalizację. W przypadku, gdy obliczone w operacji XOR q nie jest pierwsze, wtedy poszukiwana jest kolejna (większa od znalezionej) liczba pierwsza. W oryginalnym dokumencie algorytm byłby powtarzany od nowa.

Kolejnym krokiem jest generacja liczby p. Liczba p dla danej liczby q może zostać wygenerowana przy pomocy kroków 6. - 15. algorytmu przedstawionego w sekcji 2.2 załącznika drugiego do dokumentu FIPS PUB 186.

[3] `https://www.dlitz.net/software/pycrypto/`.

```
while True:                                  Powtarzamy aż do generacji poprawnego p.
    seed, q = generate_q()
    n = divmod(bits - 1, 160)[0]             Obliczamy n ze wzoru L − 1 = n ∗ 160 + b.
    counter, offset, v = 0, 2, {}
    b = q >> 5 & 15                          Obliczamy b.
    twopowbitsone = pow(2L, bits - 1)

    while counter < 4096:                    Dla danego q generujemy p (2^12 prób).
        for k in range(0, n+1):
            v[k] = bytes_to_long(SHA.new(    Obliczamy V[k] ze wzoru V[k] = [SHA(SEED+
                seed + str(offset) +         offset + k)  mod 2^g].
                str(k)).digest())
                                             Obliczamy W ze wzoru V_0 + V_1 ∗ 2^160 + ··· + (V_n
                                             mod 2^b) · 2^(n·160)...
            w = v[n] % pow(2L, b)            ... najpierw obliczając skrajnie prawy wyraz...

            for k in range(n - 1, -1, -1):   ... a następnie obliczając kolejne wyrazy idąc
                w = (w << 160L) + v[k]       w lewą stronę.

            x = w + twopowbitsone            Obliczamy X = W + 2^(L−1).

            c = x % (2 * self.q)             Obliczamy c = X  mod 2 · q.

            p = x - (c - 1)                  Obliczamy p = X − (c − 1).

            if twopowbitsone <= p and        Jeśli uzyskana liczba jest pierwsza to poprawnie
                isPrime(p):                  wygenerowaliśmy p i przerywamy algorytm.
                break

            counter += 1                     Inkrementujemy licznik.

            offset += n + 1                  Zwiększamy przesunięcie.

    if counter < 4069:                       Jeżeli liczba p jest poprawna dla danego q to kon-
        break                                tynuujemy generację klucza.
```

Generacja klucza Po wygenerowaniu liczb p oraz q przystępujemy do generacji (g, x, y).

```
while True:
    h = bytes_to_long(os.urandom(bits))      Generujemy losowe h z pierścienia modulo p − 1.
        % (p - 1)
    g = pow(h, divmod(p - 1, q)[0], p)       Obliczamy g jako h podniesione do części całko-
                                             witej z dzielenia (p−1)/q w pierścieniu modulo p.
    if 1 < h < p - 1 and g > 1:
        break                                Sprawdzamy czy g i h spełniają zadane warunki.
```

```
while True:
    x = bytes_to_long(os.urandom(20))
    if 0 < x < q:
        break
y = pow(g, x, p)
```

Generujemy losową liczbę x...

... która spełnia warunek $0 < x < q$.

Obliczamy klucz publiczny y.

Podpisywanie wiadomości Podpisywanie wiadomości zostało zaimplementowane zgodnie z założeniami algorytmu DSA.

```
def sign(self, message):
    m = bytes_to_long(SHA.new(message)
        .digest())
    k = randint(1, q - 1)
    inverse_k = inverse(k, q)
    r = pow(g, k, p) % q
    s = (inverse_k * (m + x * r)) % q
    return r, s
```

Obliczamy wartość skrótu wiadomości `message`.

Wybieramy losowe k z przedziału $(0, q)$.

Obliczamy odwrotność k w pierścieniu modulo q.

Obliczamy $r = (g^k \mod p) \mod q$.

Obliczamy $s = (k^{-1}(SHA(M) + xr)) \mod q$.

Weryfikacja podpisu Podobnie, weryfikacja podpisu również została zaimplementowana zgodnie z założeniami algorytmu DSA.

```
def verify(self, message, r, s):
    m = bytes_to_long(SHA.new(message)
        .digest())
    if not (0 < r < q) or not (0 < s < q):
        return False
    w = inverse(s, q)
    u1 = (m * w) % q
    u2 = (r * w) % q
    v = ((pow(g, u1, p) * pow(y, u2, p))
        % p) % q
    return v == r
```

Obliczamy skrót wiadomości *message*.

Jeżeli r lub s nie znajdują się w zadanych przedziałach, to podpis jest nieprawidłowy.

Obliczamy $w = s^{-1} \mod q$.

Obliczamy $u_1 oraz u_2$.

Obliczamy $v = (g^{u_1} \cdot y^{u_2} \mod p) \mod q$.

Dokonujemy weryfikacji.

Aplikacja Aplikacja została napisana w języku Python. Do uruchomienia wymagany jest interpreter języka Python w wersji 2.7.X (testowane dla 2.7.5). Program można uruchomić przechodząc do głównego katalogu projektu i wywołując z linii poleceń komendę `python main.py`.

Aplikacja pracuje w trybie interaktywnym. Użytkownik może skorzystać z trzech różnych funkcji:

1. Generacja klucza DSA. Program wygeneruje i zapisze do plików dwa klucze:
   - Klucz prywatny, który służy do podpisywania wiadomości.
   - Klucz publiczny, który służy do weryfikowania podpisu.
2. Podpisywanie wiadomości. Wykorzystanie tej funkcjonalności wymaga wygenerowania klucza prywatnego DSA. Aplikacja wykorzysta go do podpisania wiadomości wprowadzonej przez użytkownika. Podpis, wraz z wiadomością, zostaną zapisane do pliku.
3. Weryfikacja podpisu. Wykorzystanie tej funkcjonalności wymaga wygenerowania klucza publicznego DSA, a także podpisanej wiadomości. Aplikacja dokona sprawdzenia czy wiadomość została podpisana przez właściciela klucza prywatnego sparowanego z danym publicznym kluczem DSA.

Wybranie pierwszej operacji skutkuje wyświetleniem dwóch monitów do użytkownika:

- Za pierwszym razem należy wprowadzić nazwę pliku, do którego zostanie zapisany klucz prywatny.
- Za drugim razem należy wprowadzić nazwę pliku, do którego zostanie zapisany klucz publiczny.

Wybranie drugiej operacji skutkuje wyświetleniem trzech monitów do użytkownika:

- W pierwszym kroku należy wprowadzić treść wiadomości, która zostanie podpisana z wykorzystaniem algorytmu DSA.
- W drugim kroku należy wprowadzić nazwę pliku, w którym znajduje się klucz prywatny, którym zostanie podpisana wiadomość. Nazwa musi wskazywać na poprawnie wygenerowany plik klucza. W przeciwnym przypadku program zakończy się z komunikatem o błędzie.
- W trzecim kroku należy wprowadzić nazwę pliku, do którego zostanie zapisana wiadomość wraz z podpisem.

Wybranie trzeciej operacji skutkuje wyświetleniem dwóch monitów do użytkownika:

- W pierwszym kroku należy wprowadzić nazwę pliku, w którym znajduje się klucz publiczny, dla którego będziemy weryfikować podpis. Nazwa musi wskazywać na poprawnie wygenerowany plik klucza. W przeciwnym przypadku program zakończy się z komunikatem o błędzie.
- W drugim kroku należy wprowadzić nazwę pliku, w którym znajduje się podpisana wiadomość. Nazwa musi wskazywać na poprawnie wygenerowany podpis. W przeciwnym przypadku program zakończy się z komunikatem o błędzie.

W rezultacie sprawdzenia poprawności podpisu, program wyświetli jeden z dwóch komunikatów:

- `Sign is VALID!` oznacza, że podpis wiadomości został zweryfikowany pozytywnie.
- `Sign is INVALID!` oznacza, że podpis wiadomości został odrzucony.

Wszystkie dane eksportowane przez aplikację (klucze, podpis) zapisywane są w zwykłym formacie tekstowym i mogą być modyfikowane przez użytkownika w ramach testowania algorytmu.

Testy

Scenariusz testów opisany jest testami jednostkowymi klasy `DSAKey`. Znajdują się one w module `DSAKeyTests`.

Podstawowym testem na poprawne działanie algorytmu jest podpisanie wiadomości kluczem prywatnym z wykorzystaniem algorytmu DSA, a następnie zweryfikowanie podpisu tej samej wiadomości przy użyciu klucza publicznego. Opisuje to scenariusz `test_positive_workflow()`. W scenariuszu tym:

1. Generowana jest losowa para kluczy prywatny oraz publiczny.
2. Przy pomocy klucza prywatnego podpisywana jest wiadomość `Test message`.
3. Przy pomocy klucza publicznego weryfikowany jest podpis wiadomości `Test message` uzyskany w punkcie 2.
4. Do poprawnego zakończenia testu wymagane jest, aby podpis był prawidłowy.

W dalszej kolejności należy upewnić się, że jakakolwiek modyfikacja wiadomości powoduje, że podpis przestaje być ważny. Sprawdzenie takie wykonywane jest w trzech osobnych scenariuszach:

- Zmodyfikowana wiadomość posiada zmienioną literę na jednej pozycji.
- Zmodyfikowana wiadomość ma zamienioną kolejność dwóch liter.
- Zmodyfikowana wiadomość została skrócona.

Scenariusze te mają za zadanie sprawdzić, że każda drobna modyfikacja wiadomości powoduje, iż podpis traci ważność.

Pierwszy ze scenariuszy negatywnych (zmieniona litera na jednej pozycji) przebiega następująco:

1. Generowana jest losowa para kluczy prywatny oraz publiczny.
2. Przy pomocy klucza prywatnego podpisywana jest wiadomość `Test message`.
3. Przy pomocy klucza publicznego weryfikowany jest podpis wiadomości uzyskany w punkcie 2., przy czym wiadomość M zostaje zastąpiona treścią `Best message`.
4. Do poprawnego zakończenia testu wymagane jest, aby podpis został odrzucony.

Kolejny ze scenariuszy negatywnych (zamieniona kolejność dwóch liter) przebiega następująco:

1. Generowana jest losowa para kluczy prywatny oraz publiczny.
2. Przy pomocy klucza prywatnego podpisywana jest wiadomość `Test message`.
3. Przy pomocy klucza publicznego weryfikowany jest podpis wiadomości uzyskany w punkcie 2., przy czym wiadomość M zostaje zastąpiona treścią `Tset message`.
4. Do poprawnego zakończenia testu wymagane jest, aby podpis został odrzucony.

Ostatni ze scenariuszy negatywnych (skrócona wiadomość) przebiega następująco:

1. Generowana jest losowa para kluczy prywatny oraz publiczny.
2. Przy pomocy klucza prywatnego podpisywana jest wiadomość `Test message`.
3. Przy pomocy klucza publicznego weryfikowany jest podpis wiadomości uzyskany w punkcie 2., przy czym wiadomość M zostaje zastąpiona treścią `message`.
4. Do poprawnego zakończenia testu wymagane jest, aby podpis został odrzucony.

Dodatkowo należy upewnić się, że zmiana podpisu wiadomości (a więc jednej lub obu liczb z pary (r, s)) również spowoduje, że podpis zostanie odrzucony. Testowane są trzy przypadki:

- Wybierana jest losowa wartość r'.
- Wybierana jest losowa wartość s'.
- Wybierana jest losowa wartość pary (r', s').

Aby „uprawdopodobnić", że losowa wartość klucza (lub jego części) zostanie uznana za poprawną w każdym teście generowane są zmienione liczby o długości bitowej zgodnej z oryginalnymi długościami bitowymi (r, s).

Pierwszy ze scenariuszy negatywnych dla zmienionego podpisu (r, s) wiadomości przebiega następująco:

1. Generowana jest losowa para kluczy prywatny oraz publiczny.
2. Przy pomocy klucza prywatnego podpisywana jest wiadomość `Test message`.
3. Generowana jest losowa wartość r' o długości bitowej równej długości bitowej r.
4. Przy pomocy klucza publicznego weryfikowany jest podpis wiadomości `Test message` o wartości (r', s).
5. Test zakończy się poprawnie wtedy i tylko wtedy, gdy weryfikacja zakończy się tym samym wynikiem co porównanie $r' == r$.

Kolejny ze scenariuszy negatywnych dla zmienionego podpisu (r, s) wiadomości przebiega następująco:

1. Generowana jest losowa para kluczy prywatny oraz publiczny.
2. Przy pomocy klucza prywatnego podpisywana jest wiadomość `Test message`.
3. Generowana jest losowa wartość s' o długości bitowej równej długości bitowej s.
4. Przy pomocy klucza publicznego weryfikowany jest podpis wiadomości `Test message` o wartości (r, s').
5. Test zakończy się poprawnie wtedy i tylko wtedy, gdy weryfikacja zakończy się tym samym wynikiem co porównanie $s' == s$.

Ostatni ze scenariuszy negatywnych dla zmienionego podpisu (r, s) wiadomości przebiega następująco:

1. Generowana jest losowa para kluczy prywatny oraz publiczny.
2. Przy pomocy klucza prywatnego podpisywana jest wiadomość `Test message`.
3. Generowana jest losowa wartość r' o długości bitowej równej długości bitowej r.
4. Generowana jest losowa wartość s' o długości bitowej równej długości bitowej s.
5. Przy pomocy klucza publicznego weryfikowany jest podpis wiadomości `Test message` o wartości (r', s').
6. Test zakończy się poprawnie wtedy i tylko wtedy, gdy weryfikacja zakończy się tym samym wynikiem co porównanie $r' == r \wedge s' == s$.