SIKE: Los, Stop, schade!

13.10.2023

Jakob & Marvey. Workshop 2023 IT Security. Dozent: Dr. Wolf Müller

Inhaltsverzeichnis

SIKE und NIST

- National Institute of Standards and Technology
- Post Quantum Cryptography Standardization Program/Competition
- Supersingular Isogeny Key Encapsulation
  - Basiert aus SIDH Schlüssel Austausch
  - Potentielle Kandidat für PQC Program
  - Ist nicht mehr sicher, ist schnell gebrochen ohne viel Leistung

Mathematische Grundlagen

SIKE mit SageMath

- Ursprünglich wurde der Angriff mit Magma durchgeführt.
- SageMath ist OpenSource, jeder kann den Angriff probieren
- Angriff ist in Github dokumentiert
- Implementierung in SageMath ist schneller als der Magma-Angriff
- Parametern sind von NIST gegeben

SIKE Parametern

- Die Parametern sind für a und b
- Werte sind Public im Sinn des Angriffs
- `p = 2^a*3^b - 1`

```
SIKE_parameters = {
    "SIKEp434" : (216, 137),
    "SIKEp503" : (250, 159),
    "SIKEp610" : (305, 192),
    "SIKEp751" : (372, 239)
}
```

- Idee: Wenn **p** groß genug ist, kann es nicht mit Brute Force gebrochen werden
- P-> Elliptische Kurven und Isogenien

SetUp

- Elliptische Kurve ist mit SageMath generiert (E_start)
- Torsionspunkten zu die Kurve für die basis 2^a und 3^b sind generiert.
- Bobs Schlüssel wird zufällig generiert(“randint”)
- Pushing3Chain() um die isogeny chains zu erhalten
  - Alle Werten in Pushing3Chain() sind Public
- Print Anweisung für Bobs private key

```
def get_l_torsion_basis(E, l):
    n = (p+1) // l
    return (n*G for G in E.gens())

P2, Q2 = get_l_torsion_basis(E_start, 2^a)
P3, Q3 = get_l_torsion_basis(E_start, 3^b)

# Make sure Torsion points are
# generated correctly
assert 2^(a-1)*P2 != infty
assert 3^(b-1)*P3 != infty
assert P2.weil_pairing(Q2, 2^a)^(2^(a-1)) != 1
assert P3.weil_pairing(Q3, 3^b)^(3^(b-1)) != 1

# generate challenge key
Bobskey = randint(0,3^b)

EB, chain = Pushing3Chain(E_start, P3 + Bobskey*Q3, b)
# Speeds things up in Sage
EB.set_order((p+1)^2)
PB = P2
for c in chain:
    PB = c(PB)
QB = Q2
for c in chain:
    QB = c(QB)

print(f"If all goes well then the following digits should be found: {Integer(Bobskey).digits(base=3)}")
```

- Pushing3Chain(E_start, P3 + Bobskey*Q3, b)
  - E_start -> elliptische Kurve
  - P3, Q3 -> Torsionspunkte auf E_start
  - Bobskey -> zufällig generierte Int (0,3^b)
  - P3 + Bobskey*Q3 -> ein neues Punkt auf E_start
  - b -> länge der Isogeny Chain
  - letzte Kurve und Isogeny Chain generiert
- Von das Chain werden mehrere Werten verwendet für das Angriff

Beginn des Angriffs

- Skb -> Liste für Bobs Schlüsselwerten(im Ternär)
- Uvtable -> Liste mit verschiedene Parametern für Isogeny Generierung
- Erste **bet1** Werten werden getestet.
- Glue and Split Oracle

```
expdata = [[0, 0, 0] for _ in range(b-3)]
# for i in [1..b-3] do
for i in range(1,b-2):
    # if IsOdd(b-i) then
    if (b-i)%2 == 1:
        index = (b - i + 1) // 2
        exp = uvtable[index-1][1]
        if exp <= a:
            u = uvtable[index-1][2]
            v = uvtable[index-1][3]
            expdata[i-1] = [exp, u, v]

# gather digits until beta_1
bet1 = 0
while expdata[bet1][0] == 0:
    bet1 += 1
bet1 += 1

ai = expdata[bet1-1][0]
u  = expdata[bet1-1][1]
v  = expdata[bet1-1][2]

print(f"Determination of first {bet1} ternary digits. We are working with 2^{ai}-torsion.")
```

a = 216 und **b = 137** mit **uvtable** die Werten sind:
-bet1 = 2 (die erste 2 Werte von Bobs Schlüssel)
-ai = expdata[2-1][0] = 135
-u = expdata[2-1][1] = 68402650496677101758628224525965
-v = expdata[2-1][2] = 49107347770682940235806407951555 7

```
68            [ 135, 216, 24659981268679152157360132862957 7, 68402650496677101758628224525965 ],
```

-U und v für Distortionspunkte in Glue and Split Oracle

Glue and Split Oracle

- Does22ChainSplit
- Durch Richelot Isogeny Property
- Kurve C -> E_start + Transformation
- C und EB werden “geklebt”
- Richelot Curve wird versucht zu dekomponieren
- Wenn es dekomponiert -> Split gefunden!
- Die Schritte zeigen das ein Punkt erreicht wird wo Richelot Kurve zu 2 verschiedene Kurven separiert werden kann.

```
# for j in CartesianPower([0,1,2], bet1) do
for j in product([0,1,2], repeat=int(bet1)):
    print(f"Testing digits: {[j[k] for k in range(bet1)]}")

    # tauhatkernel = 3^bi*P3 + sum([3^(k-1)*j[k-1] for k in range(1,beta+1)])*3^bi*Q3
    tauhatkernel = 3^bi*P3
    for k in range(1, bet1+1):
        tauhatkernel += (3^(k-1)*j[k-1])*3^bi*Q3

    tauhatkernel_distort = u*tauhatkernel + v*two_i(tauhatkernel)

    C, tau_tilde = Pushing3Chain(E_start, tauhatkernel_distort, bet1)

    P_c = u*P2 + v*two_i(P2)
    for taut in tau_tilde:
        P_c = taut(P_c)
    Q_c = u*Q2 + v*two_i(Q2)
    for taut in tau_tilde:
        Q_c = taut(Q_c)

    # if j eq <2 : k in [1..bet1]> or Does22ChainSplit(C, EB, 2^alp*P_c, 2^alp*Q_c, 2^alp*PB, 2^alp*QB, ai) then
    if j == (2,)*bet1 or Does22ChainSplit(C, EB, 2^alp*P_c, 2^alp*Q_c, 2^alp*PB, 2^alp*QB, ai):
        print("Glue-and-split! These are most likely the secret digits.")
        for k in j:
            skB.append(k)
        break

print(skB)
```

- Die Parametern und Werten von 0 bis 2 werden getestet.
- Falls Does22ChainSplit = True, Werten -> Teil des Schlüssels.
- Angriff geht weiter bis alle Werten gefunden sind.
- Prolongationen werden gebraucht, da die Splits zu klein werden können.

Brute Force

- Die letzte 3 Werten werden mit Brute Force gefunden
- Schleife probiert Werte mit den Gleichen Parametern
- Ziel: Kurve generieren der das Isogeny Chain und Werte mit den Anfangs Kurve vergleichen kann.

```
# bridge last safety gap
tim2 = time.time()

found = false
for i in range(3^5+1):
    bobskey = key + i*3^(b-3)
    bobscurve, _ = Pushing3Chain(E_start, P3 + bobskey*Q3, b)
    if bobscurve.j_invariant() == EB.j_invariant():
        found = true
        break
```

Live Demo

Vielen Dank für Ihre Aufmerksamkeit!