The speed of digital signatures

Applied Cryptography, Project 2, 2022/2023
Universidade de Aveiro

Name: Camila Fonseca
Nmec: 97880
All code presented available at https://github.com/lnryatt/CA/tree/master/prj2

Index

Implementation

Key Generation

All the keypairs are generated via a Python script. RSA keys are generated for **1024** bit, **2048** bit and **4096** bit key sizes
All keys use the public exponent suggested in Python's cryptography.io library documentation[1], 65537.

Elliptic Curve keys are generated for the following curves:

- **NISTP** - 256,384,521
- **NISTK** - 163,283,409
- **NISTB** - 163,283,571

Key generation via python's Cryptography.io library is trivial:
RSA:

```
private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=size_key,
    )
    public_key = private_key.public_key()
```

EC:

```
    private_key = ec.generate_private_key(curve)
    public_key = private_key.public_key()
    return private_key, public_key
```

…where 'curve' is the selected elliptic curve.

Signature Performance

The methodology used to measure the performance for both algorithms was the following, repeated for every keypair available*:

- Given two integers, **n=1000** and **m=100,** the same message was signed and verified **m** times consecutively, while timing the execution of all **m** rounds, and storing the duration into an array. This process was repeated **n** times, so that in total we have an array with **n** elements, each being the time taken to sign and verify the message **m** times.
- From this array we then select the minimum value, and divide it by **m**. This value is our performance measure.

*Except in Rust, where only two of the chosen elliptic curves have been implemented.

Key Loading

The keys used are always the same, loaded from their .pem files, which is seen in the following snippets (each operation repeated for the respective public key as well.):

Python - Both:

```
with open("../keys/"+cr+"_"+key_size+"_private_key.pem", "rb") as f:
               private_key = serialization.load_pem_private_key(
                   f.read(),
                   password=None,)
```

Rust - RSA (EC is similar):

```
let priv_filename = format!("../../keys/{}_private_key.pem", size);
let private_key = RsaPrivateKey::read_pkcs1_pem_file(&priv_filename)
          .expect("Failed to read private key!");
```

Java - RSA *partial source***:**

```
KeyFactory factory = KeyFactory.getInstance("RSA");
File privkeyFile = new File("../../keys/" + key_size + "_private_key.pem");
try (FileReader keyReader = new FileReader(privkeyFile);
    PemReader pemReader = new PemReader(keyReader)) {

   PemObject pemObject = pemReader.readPemObject();
   byte[] content = pemObject.getContent();

   java.security.spec.KeySpec spec = new
java.security.spec.PKCS8EncodedKeySpec(content);
   return java.security.KeyFactory.getInstance("RSA", new
BouncyCastleProvider()).generatePrivate(spec);
}
```

Java(EC):

```
File privKeyFile = new File("../../keys/" + curve_name + "_" + key_size +
"_private_key.pem");
Security.addProvider(new org.bouncycastle.jce.
      provider.BouncyCastleProvider());

Object parsed = new org.bouncycastle.openssl.PEMParser(new FileReader
      (privKeyFile)).readObject();
KeyPair pair = new org.bouncycastle.openssl.jcajce.JcaPEMKeyConverter()
      .getKeyPair((org.bouncycastle.openssl.PEMKeyPair)parsed);
return pair.getPrivate();
```

For the three languages, it is measured how long the operations to sign and verify take.
In **rust**, the signing and verification library functions are called right away:

```
        for _ in 0..n {
            let start = std::time::SystemTime::now();
            for _ in 0..m {
                let data = b"this is a message";
                let signature = signing_key.sign_with_rng(&mut rng, data);
                verifying_key
                    .verify(data, &signature)
                    .expect("[!] - failed to verify!!");
            }
            let end = std::time::SystemTime::now();
            let duration = end.duration_since(start).expect("Time went
backwards");
            speeds.push(duration.as_nanos());
        }
```

Whereas in python and Java they are wrapped in methods:
Python - EC loop

```
            for n in range(n_size):

            start = time.time()
            for m in range(m_size):
                signature = sign_ec(message, private_key)
                ec_verify(message, signature, public_key)
            end = time.time()
            time_taken = end-start
            speeds.append(time_taken)
```

Python - EC signer wrapper (RSA is very similar)

```
def sign_ec(message, private_key):
   signature = private_key.sign(
   message,
   ec.ECDSA(hashes.SHA256())
   )
```

Java - RSA signer wrapper

```
public static byte[] GenerateSignature(String plaintext, KeyPair keys) throws
SignatureException, UnsupportedEncodingException, InvalidKeyException,
NoSuchAlgorithmException, NoSuchProviderException {

   Security.addProvider(new BouncyCastleProvider());
   Signature ecdsaSign = Signature.getInstance("SHA256withECDSA", "BC");
   ecdsaSign.initSign(keys.getPrivate());
   ecdsaSign.update(plaintext.getBytes(StandardCharsets.UTF_8));
   return ecdsaSign.sign();
}
```

References

…and consulted material.

[1] - RSA — Cryptography 39.0.0.dev1 documentation
[2] - bouncycastle-examples/Rsa.java at master
[3] - How to Read PEM File to Get Public and Private Keys | Baeldung
[4] - https://github.com/anonrig/bouncycastle-implementations
[5] - rsa - Rust
[6] - elliptic_curve - Rust

Speed Measurements (Results)

			Python	Rust	Java
RSA	**PKCS#1**	1024-bit key	0.00010318994522094727	0.000294368	0.004192233
		2048-bit key	0.0005173611640930175	0.001492722	0.005002908
		4096-bit key	0.003352668285369873	0.008831257	0.009350448
	PSS	1024-bit key	0.00010087966918945313	0.000317642	0.000182862
		2048-bit key	0.0005218625068664551	0.001500804	0.000825317
		4096-bit key	0.003411595153808595	0.00873586	0.005102905
ECDSA	NIST P-256		8.111000061035156e-05	0.000327061	0.004469753
	NIST P-384		0.0009912538528442383	0.001434693	0.005433555
	NIST P-521		0.0005058050155639648		0.006793735
	NIST K-163		0.00040699481964111327		0.005199567
	NIST K-283		0.000886080265045166		0.006866786
	NIST K-409		0.0014113998413085938		0.01502636
	NIST B-163		0.00040699481964111327		0.00509216
	NIST B-283		0.0008511447906494141		0.006642257
	NIST B-571		0.0033850979804992674		0.008967106

- All values are presented in **seconds.**

All measurements were done in a computer with the following characteristics:
Model: LG Gram 15Z90Q
OS: Pop!_OS 22.04 LTS (x86_64)
CPU: Intel i5-1240P @ 4.4GHz (16 cores)
GPU: Intel Alder Lake-P
RAM: 16GB LPDDR5@2105MHz (clock speed)
Rust and Python measurements were taken with only the terminal open, however, Java had IntelliJ open as well. Rust measurements were taken on a release/optimized executable.