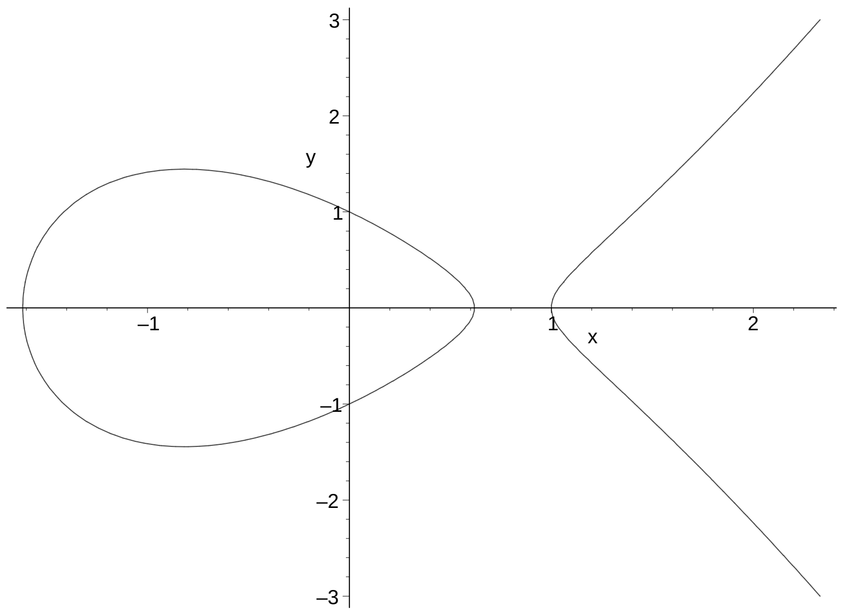
A Guide to the **ECCelerate Library** 5.0



Elliptic Curve Cryptography
by IAIK

July 12, 2018

Contents

List of Tables

Listings

List of Figures

1 Introduction

This document provides a short introduction to IAIK's ECCelerate^{TM} Java library. This library provides support for elliptic curve cryptography in Java.It was developed from scratch and replaces the old IAIK-ECC library. Moreover, it is compatible to Java versions 1.6 or higher. Please use our old ECC library, if you still need to be compatible to older Java versions.

We provide a short overview on how to use the library to create ECDSA and EdDSA digital signatures or shared secrets using the ECDH, X25519 and X448 key agreement algorithms. Elliptic curve crypto systems per se are not within the scope of this tutorial.

2 Features

The ECCelerate library allows to create and verify ECDSA and EdDSA signatures and supports ECDH, X22519 and X448 key agreement schemes. The ECCelerate library is an add-on to the IAIK JCE toolkit (of version 5.51 or higher) and therefore you have to make sure the JCE library is included in your classpath. The following list summarizes the main features of our implementation:

- Written entirely in the Java^{TM} language.
- Compliant with ANSI X9.62, IEEE 1363, FIPS 186-3, NIST SP800-57, SEC1 v2.0, SEC2 v2.0, IEEE P1363a and ANSSI.
- Incorporates ECDSA with SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHA-1, SHA224, SHA256, SHA384, SHA512 and RipeMD160.
- Incorporates EdDSA using Curve25519 and Curve448.
- Incorporates key exchange using X25519 and X448.
- Fast finite field arithmetic in prime fields.
- Fast finite field arithmetic in binary fields. In binary fields we only use polynomial base representation. This is mainly because of the patent situation, but there is no reason to use Gaussian normal bases.
- Support for elliptic curve arithmetic with affine and several types of projective coordinates (Extended Jacobian, Jacobian, Lopez-Dahab, . . .).

- Extensive set of domain parameters.
- JCA/JCE integration of ECDSA, EdDSA, ECDH, X25519, X448, ECIES and ECMQV (optional).
- ASN.1 encoding for use with X.509 and PKCS#8.
- Provides interfaces to IAIK-CMS and IAIK iSaSiLk libraries.
- Support for point compression.
- Optional (basic) protection against (remote) timing side-channels.
- Optional arithmetic speed-ups to enhance performance (Koblitz curves, various optimized scalar multipliers, ...). These are delivered with the addon due to the vague patent situation.

3 Usage

This section describes how to use the `ECCelerate` library to create and verify digital signatures using ECDSA. The integration into the JCE/JCA framework provides for an easy combination with existing systems.

3.1 Basic Usage

Like any other signature one can create ECDSA signatures via the JCA framework. Before you can use the `ECCelerate` library you have to register the security provider as described in Oracle's cryptography architecture guide. This can be done either statically, using the java.security file or dynamically, as shown below.

```java
// use either this
iaik.security.ec.provider.ECCelerate.addAsProvider();
// or this
Security.addProvider(new iaik.security.ec.provider.ECCelerate
    ());
```

Listing 1: Registering the ECCelerate-Provider

Both code lines are equivalent. After the provider registration you can easily create and validate signatures. The following code fragment shows the creation and validation of an 256 bit ECDSA signature[1].

[1]For an extensive demo, also see the *demo.ecdsa.ECDSADemo* class.

```
 1 // generating a key pair
 2 KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC");
 3 kpg.initialize(256);
 4 KeyPair kp =  kpg.generateKeyPair();
 5
 6 // creating the signature
 7 Signature s = Signature.getInstance("ECDSA");
 8 s.initSign(kp.getPrivate());
 9 s.update(SIG_DATA);
10 byte[] signature = s.sign();
11
12 // verifying the signature
13 s.initVerify(kp.getPublic());
14 s.update(SIG_DATA);
15 if (s.verify(signature)) {
16     System.out.println("Sig. validation successful");
17 } else {
18     System.out.println("Sig. validation failed");
19 }
```

Listing 2: Simple ECDSA Signature

The code is equivalent to a conventional signature (e.g. RSA or DSA) except for the Strings “EC” and “ECDSA” in line 2 and line 7, respectively. The *KeyPairGenerator* is told to generate a 256 bit EC key pair. The following table can be used to select the proper key size (compared to the security of other algorithms):

Symmetric Key Length	Example Algorithm	Prime Field $\|p\|$ *	Binary Field m
112	Triple-DES	224	233
128	AES-small	256	283
192	AES-medium	384	409
256	AES-large	521	571

* binary expansion of the integer p

Table 1: Appropriate key lengths (according to NIST SP800-57a)

When initializing the *KeyPairGenerator* via the required key size, the NIST recommended elliptic curve domain parameters will be used according to the following table:

The next section shows how you can use your desired set of domain parameters.

KeySize Parameter	Used Domain Parameter
$keysize \leq 192$	P-192
$192 < keysize \leq 224$	P-224
$224 < keysize \leq 256$	P-256
$256 < keysize \leq 384$	P-384
$384 < keysize$	P-521

Table 2: Domain parameters when initializing *KeyPairGenerator* for certain key sizes

3.2 Choosing the Domain Parameters

If you want to use other domain parameters, than those listed in Table 2 you have to initialize the *KeyPairGenerator* with an *AlgorithmParamerSpec* object. You can use the *ECStandardizedParameterFactory* class to select a certain set of domain parameters. Listing 3 shows, how to create a key pair using the NIST domain parameters B-233.

```
ECParameterSpec parameter = ECStandardizedParameterFactory.
    getParametersByName("B-233");
KeyPairGenerator kpg = KeyPairGenerator.getInstance("ECDSA");
kpg.initialize(parameter);
KeyPair kp = kpg.generateKeyPair();
```

Listing 3: ECDSA Signature over B-233

The *ECStandardizedParameterFactory* implementation contains pre-defined domain parameters. All parameters can either be referenced by their names or by their Object Identifier OID.

3.3 Deterministic Signing (RFC 6979)

The ephemeral key generation processed defined in RFC 6979 to enable deterministic signing process, the parameter *DeterministicSigning* can be set per *Signature* instance. Listing 4 details how to create signatures following RFC 6979.

```
// generating a key pair
KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC");
kpg.initialize(256);
KeyPair kp =  kpg.generateKeyPair();

// creating the signature
Signature s = Signature.getInstance("ECDSA");
// enable deterministic signing
```

```java
s.setParameter(new DeterministicSigning());
s.initSign(kp.getPrivate());
s.update(SIG_DATA);
byte[] signature = s.sign();

// verifying the signature
s.initVerify(kp.getPublic());
s.update(SIG_DATA);
if (s.verify(signature)) {
    System.out.println("Sig. validation successful");
} else {
    System.out.println("Sig. validation failed");
}
```

Listing 4: Deterministic ECDSA Signature

3.4 Installing and Enabling the Addon

We incorporated a series of arithmetic speed-ups for elliptic curves over binary curves and prime curves. The patent situation is not clear for the used algorithms, so we adopted the same strategy as was done by the old IAIK-ECC library, namely, we moved the critical classes into the separate `iaik_eccelerate_addon.jar` file to let the developer decide, whether or not to include the `.jar` file. This file contains the arithmetic speed-ups. It suffices to add `iaik_eccelerate_addon.jar` to the classpath. It will then be loaded automatically.

We introduced some additional methods to configure the way the additional .jar file is used (cf. also Listing 1):

```java
// Add provider and enable the use of speed-ups (while
    disabling the classloader to search for them)
Security.addProvider(new iaik.security.ec.provider.ECCelerate
    (true));
```

Listing 5: Registering the ECCelerate Provider with the ECCelerate-Addon

Also, see `demo.ECCelerateAddonDemo` for more details. If the `ECCelerate` provider is registered as in Listing 1 and faster arithmetics for a curve is available in the addon, the library will use a classloader to see if there is an implementing class in the classpath.

Since there are cases, when the invocation of a classloader is not preferred, the `ECCelerate` provider can also be registered as in Listing 5. Then, the user has to ensure the presence of the `iaik_eccelerate_addon.jar` in the classpath because no classloader will be used. Consequently, if the `iaik_eccelerate_addon.jar` is missing, you will get an exception. If you do not require or do not want to use the addon due to the vague patent situation, you must ensure that `iaik_eccelerate_addon.jar` is not placed within the same directory as `iaik_eccelerate.jar`. Furthermore, you must either replace *true* by *false* in Listing 5 or use the provider's default constructor. Then, no speed-ups are used and the classloader is never invoked to search for them.

3.5 Using ECCelerate with the IAIK-CMS and IAIK iSaSiLk Libraries

In order to take advantage of ECC operations in context of CMS and SSL, just include the respective `.jar` file (either `iaik_eccelerate_cms.jar` or `iaik_eccelerate_ssl.jar`), which is being shipped with `ECCelerate` into the classpath. Then, execute one of the subsequent code snippets to register `ECCelerate` with the respective library:

```
iaik.cms.SecurityProvider.setSecurityProvider(new iaik.cms.
    ecc.ECCelerateProvider());
```

Listing 6: Registering the ECCelerate Provider with the IAIK-CMS library

```
iaik.security.ssl.SecurityProvider.setSecurityProvider(new
    iaik.security.ssl.ECCelerateProvider());
```

Listing 7: Registering the ECCelerate Provider with the IAIK iSaSiLk library

4 X.509 and PKCS#8

All generated signatures and public keys support the required ASN.1 encoding to be used in X.509 certificates. Private keys can be exported in PKCS#8 format. In other words, you can use the EC keys like any other key and put it into a PKCS#12 file for instance. The various demos provided with our ECCelerate library show how this can be done.

5 Elliptic Curve DH

This library supports the Discrete Logarithm Secret Value Derivation Primitive, Diffie-Hellman version, compliant with IEEE P1363a (with and without cofactor multiplication). You have to make sure that the used keys have the same domain parameters. ECDH can be used like any other key agreement scheme. The following code shows how you can derive a shared secret from already generated key pairs *kp1* and *kp2* (see Listing 2):

```
1 KeyAgreement ka = KeyAgreement.getInstance("ECDH");
2 ka.init(kp1.getPrivate());
3 ka.doPhase(kp2.getPublic(), true);
4 /* use the next line if you want to generate
5 a shared AES secret key */
6 //SecretKey sk = ka.generateSecret("AES");
7 byte[] sharedSecret1 = ka.generateSecret();
```

Listing 8: Elliptic Curve DH

If you want to use the ECDH with co-factor multiplication, you have to substitute *KeyAgreement.getInstance("ECDHwithCofactor");* in line 2 of Listing 8.

6 Elliptic Curve Integrated Encryption Scheme

`ECCelerate` includes the hybrid encryption scheme ECIES as specified by SEC1 v2.0 standard. For a complete list of supported features, we refer the reader to the Javadoc of the class `iaik.security.ec.ecies.ECIES`.

The subsequent listing illustrates how the ECCelerate ECIES implementation can typically be used (see also *demo.ecies.ECIESDemo*).

```
// select the requested curve domain parameters
ECParameterSpec params = ECStandardizedParameterFactory.
    getParametersByBitLength(keyLength);
// obtain an EC key pair generator
KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC",
    ECCelerate.getInstance());
// initialize the key pair generator with the domain
    parameters
kpg.initialize(params, random_);
// generate a key pair
KeyPair keyPair = kpg.generateKeyPair();
// create ECIES parameter with the given parameters
ECIESParameterSpec eciesParams = new ECIESParameterSpec(
    kdfParams, cipherName, macName);
// instantiate the ECIES cipher
Cipher cipher = Cipher.getInstance(algorithm, ECCelerate.
    getInstance());

/* --- ENCRYPT --- */
// initialize the cipher for encryption
cipher.init(Cipher.ENCRYPT_MODE, keyPair.getPublic(),
    eciesParams);
// perform encryption
cipher.update(CLEAR_TEXT);
```

```
byte[] cipherText = cipher.doFinal();

/* --- EN/DECODE parameters to/from ASN1 --- */
// ECIES parameter as ASN1Object
ASN1Object eciesParamsAsASN1 = eciesParams.toASN1Object();
// decode ASN1Object back to ECIES parameter
ECIESParameterSpec eciesParamsFromASN1 = ECIESParameterSpec.
    decode(eciesParamsAsASN1);

/* --- DECRYPT --- */
// initialize the cipher for decryption
cipher.init(Cipher.DECRYPT_MODE, keyPair.getPrivate(),
    eciesParamsFromASN1);
// perform decryption
cipher.update(cipherText);

byte[] encodedClearText = cipher.doFinal();
```

Listing 9: Elliptic Curve IES

7 Elliptic Curve MQV

The ECMQV protocol is an authenticated DH style key-agreement protocol. `ECCelerate` offers an implementation of the ECMQV protocol that is compliant with SEC1 v2.0. Due to the vague patent situation, however, this protocol is only delivered with the ECCelerate addon.

Just like in case of ECDH, you have to make sure that the used keys have the same domain parameters. ECMQV can be used like any other key agreement scheme. The following code shows how you can derive a shared secret using the ECMQV key agreement scheme (see also *demo.ecdsa.ECDSADemo* class) using two key pairs kA and kB:

```
// the key agreement instance of Alice
KeyAgreement kA = KeyAgreement.getInstance("ECMQV");
// the key agreement instance of Bob
KeyAgreement kB = KeyAgreement.getInstance("ECMQV");

// initialize Alice's instance with her private key
kA.init(kpA.getPrivate(), ecdhParams, random_);
// get Alice's ephemeral public key
// Note: it's crucial that no key is provided here.
Key ephemeralPublicKeyA = kA.doPhase(null, false);

// initialize Bob's instance with his private key
kB.init(kpB.getPrivate(), ecdhParams, random_);
// get Bob's ephemeral public key
// Note: it's crucial that no key is provided here.
Key ephemeralPublicKeyB = kB.doPhase(null, false);

// do the first real phase using Bob's
// ephemeral public key
kA.doPhase(ephemeralPublicKeyB, false);
// derive the secret using Bob's public key
kA.doPhase(kpB.getPublic(), true);

// derive the shared secret for Alice
String sharedSecretAString;

if (kdfParams == null) {
  // generate a plain shared secret
  final byte[] sharedSecretA = kA.generateSecret();
  sharedSecretAString = Util.toString(sharedSecretA);
} else {
  // use the KDF
  final SecretKey keyA = kA.generateSecret(cipher);
  sharedSecretAString = keyA.toString();
}
```

Listing 10: Elliptic Curve MQV

8 EdDSA Signatures

This library also supports the EdDSA signature scheme using curves Curve25519 and Curve448. Signatures can be created and verified similar to ECDSA signatures as demonstrated in Listing 2. The following listening demonstrates how to create EdDSA25519 signatures:

```
EdParameterSpec params = EdStandardizedParameterFactory.
    getParametersByName("Ed25519");
// generating a key pair
KeyPairGenerator kpg = KeyPairGenerator.getInstance("
    EdDSA25519");
kpg.initialize(params, random_);
KeyPair kp = kpg.generateKeyPair();

// creating the signature
Signature s = Signature.getInstance("EdDSA25519");
s.initSign(kp.getPrivate());
s.update(SIG_DATA);
byte[] signature = s.sign();

// verifying the signature
s.initVerify(kp.getPublic());
```

```
s.update(SIG_DATA);
if (s.verify(signature)) {
    System.out.println("Sig. validation successful");
} else {
    System.out.println("Sig. validation failed");
}
```

Listing 11: EdDSA25519 signature

For EdDSA448 signatures replace every occurrence of 25519 with 448.

9 X25519 and X448

This library supports the X25519 and X448 key exchange algorithms. The following code shows how you can derive a shared secret from ephemeral keys using X25519 (for X448 replace every occurrence of X25519 with X448):

```
EdParameterSpec params = EdStandardizedParameterFactory.
    getParametersByName("X25519");
KeyPairGenerator kpg = KeyPairGenerator.getInstance("X25519")
    ;
kpg.initialize(params, random_);

// the key agreement instance of Alice
KeyAgreement kA = KeyAgreement.getInstance("X25519");
// the key agreement instance of Bob
KeyAgreement kB = KeyAgreement.getInstance("X25519");

// the key pair of Alice
final KeyPair kpA = kpg.generateKeyPair();
// the key pair of Bob
final KeyPair kpB = kpg.generateKeyPair();

// initialize Alice's instance with her private key
kA.init(kpA.getPrivate(), params, random_);
// derive the secret using Bob's public key
kA.doPhase(kpB.getPublic(), true);

// derive the shared secret for Alice
final byte[] sharedSecretA = kA.generateSecret();
```

Listing 12: X25519 key agreement

10 Customization

This section describes some settings a user might want to change.

- Configure Encoding: The standards offer two variants on how to encode domain parameters. You may either specify the object identifier OID (if it exists) or you may encode the parameters explicitly. IAIK's ECCelerate library supports both methods. The default settings encode the parameters via their OIDs. You can change this behaviour with the following line of code.

```
ECParameterSpec.setDefaultOIDEncoding(true);
```

Listing 13: Parameter Encoding

- Public Key Verification: The library uses a plugable octet string to point conversation module as defined in ANSI X9.62. If a certificate or any other ASN.1 encoded public key is parsed, the default implementation can optionally check if this decoded point actually is an element of the elliptic curve group of interest. On default this check is disabled, to provide optimal performance. You can easily enable it with the following code line:

```
ECPublicKey.setFullCheckEnabled(true);
```

Listing 14: Point Verification

- Point compression: By default, our ECCelerate library uses the compressed form where possible to export public keys. In contrast to older versions of ECCelerate ($\leq$ 2.21), point compression is no longer delivered exclusively with the ECCelerate addon, since the corresponding patent has expired. Listing 15 shows how to disable the point compression for data export.

```
PointEncoders.setDefaultPointEncoder(PointEncoders.
    UNCOMPRESSED);
```

Listing 15: Enabling Point Compression

- Speed/Memory Tradeoff: ECCelerate offers different optimization levels in order to perform well on memory-constrained devices as well as on desktop and server environments. The speed/memory tradeoff should be set directly after adding the provider and can be done as follows:

```
ECCelerate.setOptimizationLevel(OptimizationLevel.
    FULL_SPEED);
```

Listing 16: Setting the Speed/Memory Tradeoff

Optimization Level	Description
`LIMITED_MEMORY`	Optimizes the arithmetical routines to save memory.
`MEMORY`	Optimizes the arithmetical routines so that a reasonable speed level is achieved and the memory consumption is kept low.
`DEFAULT`	Optimizes the arithmetical routines so that a good speed level is achieved and the memory consumption is kept moderate.
`SPEED`	Optimizes the arithmetical routines for speed using a larger amount of memory.
`IMPROVED_SPEED`	Aggressively optimizes the arithmetical routines for speed using a large amount of memory. Note: this option results in longer startup times.
`FULL_SPEED`	Aggressively optimizes the arithmetical routines for best speed using a large amount of memory. Note: this option results in longer startup times.

Table 3: Optimization levels

Higher optimization levels require more pre-computation time and space, but allow the library to create ECDSA signatures significantly faster (up to 2.5 times). Hence, these optimization levels are especially suited for servers. Table 3 lists the different optimization levels.

- (Basic) Timing-Attack Protection: ECCelerate allows for (basic) protection against timing side-channels (achieved by blinding secret keys). Since these countermeasures come at the expense of reduced performance, they have to be enabled on purpose. Enabling the protection is particularly important for environments which are subject to remote timing attacks apply (such as servers answering signature requests). This can be achieved as follows:

```
ECCelerate.enableSideChannelProtection(true);
```

Listing 17: Enabling Timing-Attack Protection

- Security Strength Checks: ECCelerate can be instructed to automatically check whether the combination of algorithms and parameters involved in ECDSA satisfies a minimum security strength. By default, however, this behavior has been turned off since release 2.0. In case, you want these checks enabled run in your setup:

```
ECCelerate.enforceSP80057Recommendations(true);
```

Listing 18: Security Strength Checks

A Example

The following Java program shows the creation of a 256 bit EC key pair. This first step may take some time because a secure random number generator has to be initialized (a seed has to be generated). All subsequent key pairs will be generated much faster. lines 21 to 24 show the signature creation and the rest of the code demonstrates how this signature can be validated. For more sophisticated examples please have a look at the demos provided with our library.

```
 1 import java.security.KeyPair;
 2 import java.security.KeyPairGenerator;
 3 import java.security.Signature;
 4
 5 import iaik.security.ec.provider.ECCelerate;
 6 import iaik.security.provider.IAIK;
 7
 8 public class SimpleDemo {
 9   public static void main(String[] args) {
10     // registering providers
11     IAIK.addAsrovider();
12     ECCelerate.addAsProvider();
13     byte[] data = "Data to be signed".getBytes();
14     try {
15       // generating key pair
16       KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC
            ");
17       kpg.initialize(256);
18       KeyPair kp = kpg.generateKeyPair();
19
20       // signing data
21       Signature sig = Signature.getInstance("ECDSA");
22       sig.initSign(kp.getPrivate());
23       sig.update(data);
24       byte[] signatureValue = sig.sign();
25
26       // validating signature
27       sig.initVerify(kp.getPublic());
28       sig.update(data);
29       if (sig.verify(signatureValue)) {
30         System.out.println("Sig. successfully validated!");
31       } else {
32         System.out.println("Sig. validation failed!");
33       }
34     } catch (Exception e) {
35       e.printStackTrace();
36     }
37   }
}
```

Listing 19: A Simple Demo

B Performance Charts

The subsequent performance charts show the benchmark results of our new ECCelerate library in comparison to the results of our old IAIK-ECC library. Arithmetical speedups were enabled in both cases by including the corresponding addons. Table 4 shows the hardware and software configuration, Table 5 the combinations of ECDSA with NIST curves that were used for running the benchmarks.

CPU	Intel Core i7-4790 (3.60 GHz)
Memory	16 GB DDR3
OS	Ubuntu Linux 16.04 (64 bit)
Java	OpenJDK 1.8.0_91 (64 bit, server mode)

Table 4: Hardware and software configuration

Algorithm	$E/\mathbb{F}_p$	$E/\mathbb{F}_{2^m}$
SHA1withECDSA	P-192	K-163
SHA224withECDSA	P-224	K-233
SHA256withECDSA	P-256	K-283
SHA384withECDSA	P-384	K-409
SHA512withECDSA	P-521	K-571

Table 5: Benchmarked algorithms and curves

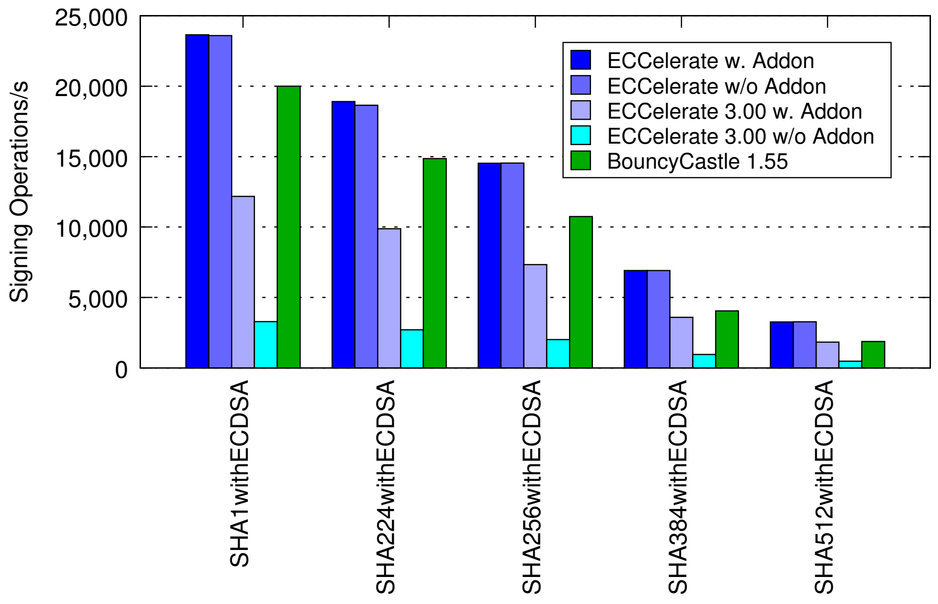


Figure 1: ECDSA Signing Operations/s using NIST Curves over $\mathbb{F}_p$

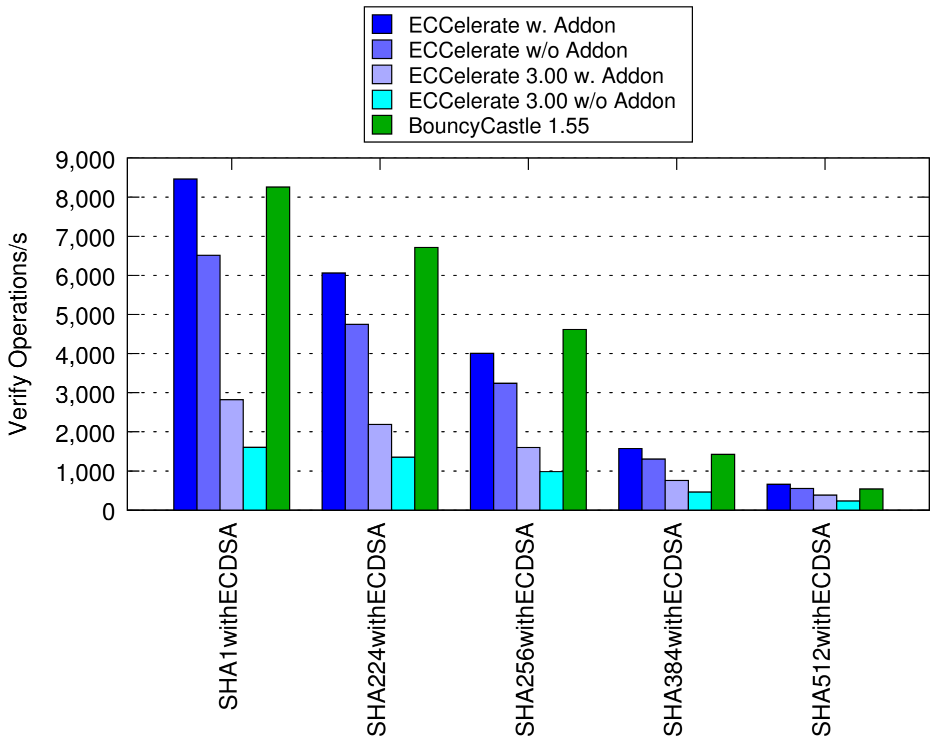


Figure 2: ECDSA Verify Operations/s using NIST Curves over $\mathbb{F}_p$

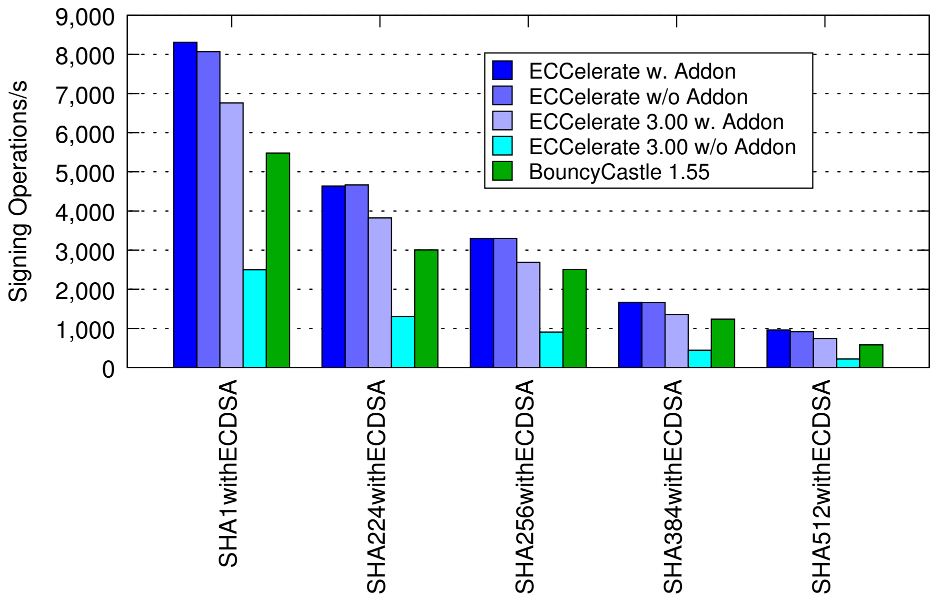


Figure 3: ECDSA Signing Operations/s using NIST Curves over $\mathbb{F}_{2^m}$

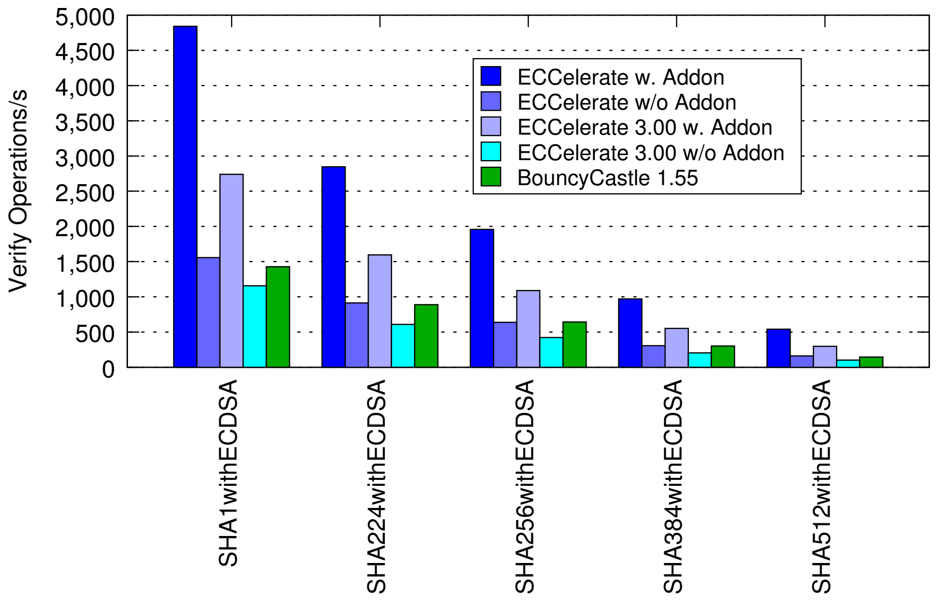


Figure 4: ECDSA Verify Operations/s using NIST Curves over $\mathbb{F}_{2^m}$