Verification-Friendly ECDSA

(and use with JOSE/COSE)

René Struik

Struik Security Consultancy

E-mail: rstruik.ext@gmail.com

IETF 112 – virtual Madrid, USA, November 2021

Outline

1. ECC Signature Schemes:
   – ECDSA Signing and Verification
   – Speed-ups with ECDSA*
2. Putting ECDSA and ECDSA* into same format
   – JWS Example
   – General Approach for Reusing ECDSA Standard
3. Transitioning Considerations
4. Conclusions, next steps

ECDSA Algorithm

ECDSA:

Signature: $\underline{r} \,||\, \underline{s}$ in most-significant-bit/octet first order

Signing equation: $e = s \cdot k - d \cdot r$ (**mod** n), where e=Hash(m), R=$k\,G$, $R \rightarrow r$

Verification: Compute $R' = (e/s)\,G + (r/s)\,Q$, where $Q= d\,G$;
check that $R' \rightarrow r$

Example: ECDSA, w/ P-256 and SHA-256 (FIPS 186-4, ANSI X9.62, etc.)

ECDSA*:

Signature: $\underline{R} \,||\, \underline{s}$ in most-significant-bit/octet first order

Signing equation: $e = s \cdot k - d \cdot r$ (**mod** n), where e=Hash(m), R=$k\,G$, $R \rightarrow r$

Verification: compute $\underline{R} \rightarrow r$;
compute $\underline{R} \rightarrow R$
check that $R = (e/s)\,G + (r/s)\,Q$, where $Q= d\,G$

ECDSA and ECDSA* have same security, but different formats

ECDSA* allows faster verification

speed-ups: ~1.3x make scalars small, which halves ECC doubles (single verify)
up to ~ 6x amortize ECC doubles and common terms (batch verify)

(This uses alternative verification equation: $\lambda\,(- R + (e/s)\,G + (r/s)\,Q) = O$ *for any* $\lambda \neq 0$)

ECDSA Algorithm

<u>ECDSA:</u>

Signature:	<u>r</u> || <u>s</u> in most-significant-bit/octet first order
Signing equation:	$e = s \cdot k - d \cdot r$ (**mod** n), where e=Hash(m), R=k G, $R \rightarrow r$
Verification:	Compute $R' = (e/s)\, G + (r/s)\, Q$, where $Q = d\, G$; check that $R' \rightarrow r$
Example:	ECDSA, w/ P-256 and SHA-256 (FIPS 186-4, ANSI X9.62, etc.)

<u>ECDSA*:</u>

Signature:	<u>R</u> || <u>s</u> in most-significant-bit/octet first order
Signing equation:	$e = s \cdot k - d \cdot r$ (**mod** n), where e=Hash(m), R=k G, $R \rightarrow r$
Verification:	compute <u>R</u> $\rightarrow r$; compute <u>R</u> $\rightarrow R$ check that $R = (e/s)\, G + (r/s)\, Q$, where $Q = d\, G$

ECDSA and ECDSA* have same security~~, but~~ and ~~different~~ same formats (with a trick)

(*our examples assume prime-order curves*)

ECDSA* allows faster verification

speed-ups: ~1.3x make scalars small, which <u>halves</u> ECC doubles (single verify)

up to ~ 6x amortize ECC doubles and common terms (batch verify)

(This uses alternative verification equation: $\lambda\,(-R + (e/s)\, G + (r/s)\, Q) = O$ *for any* $\lambda \neq 0$)

JWS Example

(RFC 7515, Appendix A.3)

JWS Protected Header:

```
{"alg":"ES256"}
```

JWS Payload:

```
{"iss":"joe",
"exp":1300819380,
"http://example.com/is_root":true}
```

JWK Key:

```
{"kty":"EC",
"crv":"P-256",
"x":"f83OJ3D2xF1Bg8vub9tLe1gHMzV76e8Tus9uPHvRVEU",
"y":"x_FEzRu9m36HLN_tue659LNpXW6pCyStikYjKIWI5a0"}
```

ECDSA signature (*r*, ± *s*):

r: 0x0ed12153 79636c48 3c2f7f15 5807d402 a3b22803 3af97c7e 17819ac3 169ea665

+*s*: 0xc50a07d3 8c3c70e5 d8f12daf 084a5480 a66590c5 f293509a 8f3f7f8a 83a354d5

-*s*: 0x3af5f82b 73c38f1b 270ed250 f7b5ab7f 168169e7 b4844dea 647a4b38 78bfd07c

*with ECDSA (**r**, +**s**)…*

*with ECDSA (**r**, -**s**)…*

Option #1: with ECDSA signature (***r***,+***s***)

eyJhbGciOiJFUzI1NiJ9.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ.
DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q

Option #2: with ECDSA signature (***r***,-***s***)

eyJhbGciOiJFUzI1NiJ9.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ.
DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmU69fgrc8OPGycO01D3tat_FoFp57SETepkeks4eL_QfA

JWS Example

(RFC 7515, Appendix A.3)

JWS Protected Header:

```
{"alg":"ES256"}
```

JWS Payload:

```
{"iss":"joe",
"exp":1300819380,
"http://example.com/is_root":true}
```

JWK Key:

```
{"kty":"EC",
"crv":"P-256",
"x":"f83OJ3D2xF1Bg8vub9tLe1gHMzV76e8Tus9uPHvRVEU",
"y":"x_FEzRu9m36HLN_tue659LNpXW6pCyStikYjKIWI5a0"}
```

ECDSA signature (*r*, ± *s*):

r: 0x0ed12153 79636c48 3c2f7f15 5807d402 a3b22803 3af97c7e 17819ac3 169ea665

+*s*: 0xc50a07d3 8c3c70e5 d8f12daf 084a5480 a66590c5 f293509a 8f3f7f8a 83a354d5

−*s*: 0x3af5f82b 73c38f1b 270ed250 f7b5ab7f 168169e7 b4844dea 647a4b38 78bfd07c

with ECDSA (*r*, +*s*)…

with ECDSA (*r*, −*s*)…

Option #1: with ECDSA signature (*r*,+*s*)

R:=(*xR*,*yR*), with *yR* odd

eyJhbGciOiJFUzI1NiJ9.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ.
DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q

Option #2: with ECDSA signature (*r*,-*s*)

R:=(*xR*,*yR*), with *yR* even

eyJhbGciOiJFUzI1NiJ9.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ.
DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmU69fgrc8OPGycO01D3tat_FoFp57SETepkeks4eL_QfA

JWS Example

(RFC 7515, Appendix A.3)

JWS Protected Header:

```
{"alg":"ES256"}
```

JWS Payload:

```
{"iss":"joe",
"exp":1300819380,
"http://example.com/is_root":true}
```

JWK Key:

```
{"kty":"EC",
"crv":"P-256",
"x":"f83OJ3D2xF1Bg8vub9tLe1gHMzV76e8Tus9uPHvRVEU",
"y":"x_FEzRu9m36HLN_tue659LNpXW6pCyStikYjKIWI5a0"}
```

ECDSA signature ($r, \pm s$):

r: 0x0ed12153 79636c48 3c2f7f15 5807d402 a3b22803 3af97c7e 17819ac3 169ea665

$+s$: 0xc50a07d3 8c3c70e5 d8f12daf 084a5480 a66590c5 f293509a 8f3f7f8a 83a354d5

$-s$: 0x3af5f82b 73c38f1b 270ed250 f7b5ab7f 168169e7 b4844dea 647a4b38 78bfd07c

with ECDSA (r, $-s$)…

Rule: Choose ECDSA option ($r, \pm s$) such that R:=(xR, yR) with yR even

Option #2: with ECDSA signature (r,$-s$)

R:=(xR,yR), with yR even, so $r \rightarrow R$ unique (r=xR in practice)

eyJhbGciOiJFUzI1NiJ9.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ.
DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmU69fgrc8OPGycO01D3tat_FoFp57SETepkeks4eL_QfA

ECDSA Algorithm

ECDSA:

Signature: $\underline{r} \,||\, \underline{s}$ in most-significant-bit/octet first order

Signing equation: $e = s \cdot k - d \cdot r$ (**mod** n), where e=Hash(m), R=k G, $R \rightarrow r$

Verification: Compute $R' = (e/s)\, G + (r/s)\, Q$, where $Q = d\, G$;
check that $R' \rightarrow r$

Example: ECDSA, w/ P-256 and SHA-256 (FIPS 186-4, ANSI X9.62, etc.)

ECDSA*:

Signature: $\underline{R} \,||\, \underline{s}$ in most-significant-bit/octet first order

Signing equation: $e = s \cdot k - d \cdot r$ (**mod** n), where e=Hash(m), R=k G, $R \rightarrow r$

Verification: compute $\underline{R} \rightarrow r$;
compute $\underline{R} \rightarrow R$
check that $R = (e/s)\, G + (r/s)\, Q$, where $Q = d\, G$

ECDSA and ECDSA* have same security~~, but~~ and ~~different~~ same formats (with a trick)

ECDSA via modified ECDSA signing procedure*

– Step 1: Generate ECDSA signature (r, s) of message m, as usual;

– Step 2: Change (r, s) to (r,-s) if ephemeral key R has y-coordinate with odd parity

ECDSA and ECDSA* have same format, since $R \rightarrow r$ map reversible ($r \rightarrow \pm R$ unique in practice)

Implementation of ECDSA*

ECDSA via modified ECDSA signing procedure*

– Step 1: Generate ECDSA signature (r, s) of message m, as usual;

– Step 2: Change (r, s) to (r,-s) if ephemeral key R has y-coordinate with odd parity

Notes:

– If (r, s) is a valid ECDSA signature, then so is (r, -s) –– the so-called malleability

This modified ECDSA signature is still an ECDSA signature, so no security impact

– Any party can perform Step 2, since for valid signatures $R := (e/s)\ G + (r/s)\ Q$

This party does not have to be the signer and this can be done retroactively

Impact on ECDSA verifiers:

– If verifier *knows* that modified signing procedure was used, $R' \rightarrow r$ has unique preimage in practice for all prime-order curves (implicit point compression R) and speed-ups always work

– If verifier does not know that modified signing procedure was used, it can still verify ECDSA signatures as usual (but will not get single-verify and batch verification speed-ups)

– If verifier wishes to use ordinary ECDSA signature verification for whatever reason, it can do so

Transitioning towards ECDSA*

Applications with IETF protocols:

Everywhere, e.g., PKIX, CMS, Certificate Transparency, OpenPGP, COSE, JOSE, lake, etc.

JOSE:

– Define new `"alg"` field named `"ES256vf"` (ES256 with modified signing)

– Verifier who does not want to use speed-ups interprets this as `"ES256"`
  This is possible since ECDSA signatures are also ECDSA signatures*

– Signer can use modified ECDSA signing procedure
  Use of new label does not allow retroactively putting "ES256"-based JWS signatures into ECDSA format, since the new label is part of the "to-be-signed data"*
  Note: this would be possible, if one would replace "ES256vf" by "ES256" before signing and verification, but this likely would create JWS processing anomalies

Alternatives:

– Put all existing ECDSA signatures retroactively into ECDSA* format ("Big Bang");

– Mandate in specific protocols.

Question is whether this would entice implementors enough to switch to always-on ECDSA*. This also comes down to side-information re whether the modified signing procedure was followed that is not part of the JWS string (something JSON parsers may not be able to handle)

Conclusions & Question to Group

Summary:

– ECDSA verification can take advantage of ECDSA* speed-ups, similarly to EdDSA, both in single-verify and batch-verify cases

– Techniques useful in *all* settings, where use of the modified ECDSA signing procedure can be signalled

– Techniques known since 2005 (single-verify), 1995 (batch-verify), published in well-respected, peer-reviewed crypto conferences. {Also used in Bitcoin signing}

Why now, why here?

– techniques known for long; standardization incentive for implementation needed

– e-health initiatives (e.g., SmartHealth, IATA Pass (jws); EU digital covid certs (cose))

Question to Group:

– Does group support this work, once crypto review gating hurdle taken?

– How can we make this quick project, so that it can have real-life impact? (e.g., already reserving code points, so that this does not become a dead horse)

– Any volunteers for implementation, getting this done asap, etc.?

– Any pointers to iana templates, other than Section 9 of RFC 7515?