Bitcoin:
Programming the Future of Money

Topics in Computer Science - ITCS 4010/5010, Spring 2025

Dr. Christian Kümmerle

Lecture 13

Encodings and Serialization

Slides are adapted from:
- "Programming Bitcoin: Learn How to Program Bitcoin from Scratch. (Jimmy Song), 1st Edition, O'Reilly, 2019.

UNIVERSITY OF NORTH CAROLINA CHARLOTTE

RECAP

Last class:

Two signature schemes based on elliptic curve arithmetic over finite fields.

- **Elliptic Curve Digital Signature Algorithm (ECDSA)**
- **Schnorr Signatures**

RECAP

- **Elliptic Curve Digital Signature Algorithm (ECDSA)**
  - Concept proposed by Neal Koblitz and Victor S. Miller in 1985
  - Standardized in 2000 by NIST
  - Used in Bitcoin since 2009, was freely available
  - Used by all address formats before Taproot upgrade

- **Schnorr Signatures:**
  - Proposed and **patented** by Claus-Peter Schnorr in 1990
  - Has certain advantages over ECDSA (will see later) and simpler
  - Patent expired in 2010, so not available at inception of Bitcoin
  - Implemented in address format introduced by 2021 Taproot upgrade

ECDSA VS. SCHNORR SIGNATURES IN BITCOIN

- **Elliptic Curve Digital Signature Algorithm (ECDSA)**
  - Public Keys (typically) serialized with 33 bytes (Compressed SEC format)
  - Signatures serialized with up to 73 bytes (Distinguished Encoding Rules, DER)

- **Schnorr Signatures:**
  - Public Keys serialized with 32 bytes (x-coordinate of elliptic curve point, choose y to be even)
  - Signatures serialized with 64 bytes

ECDSA VS. SCHNORR SIGNATURES IN BITCOIN

- **Advantages of Schnorr Signatures over ECDSA**
  - Shorter serializations (leading to memory savings)
  - Provable security guarantees based on:
    ✦Difficulty of discrete logrithm problem for the secp256k1 elliptic curve, and
    ✦Appropriateness of "Random Oracle Model" for SHA-256 hash function
  - Linearity of signatures
  - Efficient batch verification
  - Ability for "Scriptless" Multi-Signature Schemes (not standard yet)

assumes that the output for each input of hash function is uniformly randomly distributed. Not correct due to deterministic nature of hash function, but good approximation.

e.g.: ▷ 2-of-2 multisig

▷ 3-of-5 multisig

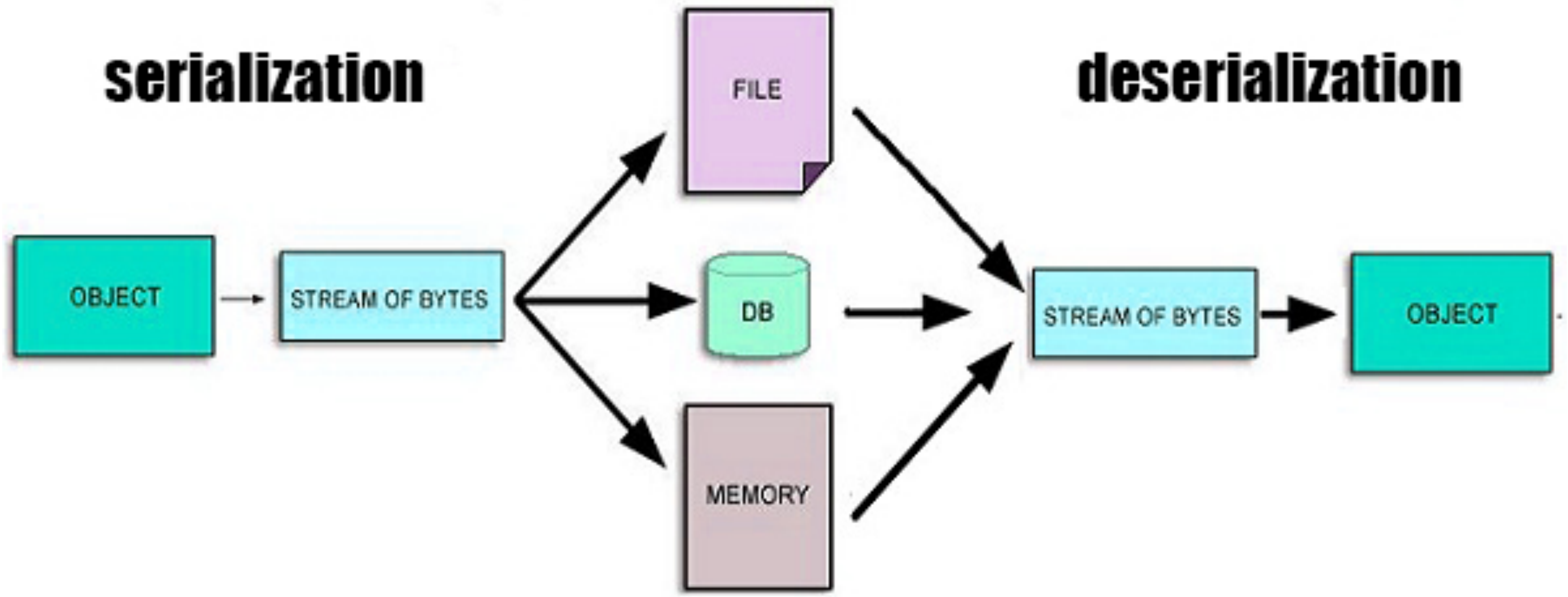
min. of parties that need to sign

max. of parties that need to sign

Serialization

SERIALIZATION

In computing, **serialization** (or **serialisation**) is the process of translating a data structure or object state into a format that can be stored (e.g. files in secondary storage devices, data buffers in primary storage devices) or transmitted (e.g. data streams over computer networks) and reconstructed later (possibly in a different computer environment).[1] When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object. For many complex objects, such as those that make extensive use of references, this process is not straightforward. Serialization of objects does not include any of their associated methods with which they were previously linked.



ENCODINGS

Integer representation of number 6:

$$[6]_{10} = [110]_2 = 1\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 = 1\cdot 4 + 1\cdot 2 = 6$$

number 6 in base 10

3 bit

$$[255]_{10} = [11111111]_2 = 256 - 1 = 2^8 - 1$$

8 bit

Hexadecimal representation:

0 1 2 3 4 5 6 7 8 9 A B C D E F

14 15

$$[255]_{10} = [\underbrace{FF}_{\text{2 hex digits}}]_{16} = 15\cdot 16 + 15$$

$$\Rightarrow \boxed{\text{Number of bytes } n \mathrel{\hat{=}} \text{Number of hex digits } 2n}$$

ENCODINGS

Example:

$\triangleright \quad 1000$

$$1000 = \underbrace{3 \cdot 256}_{768} + \underbrace{232}_{=14 \cdot 16 + 8}$$

$$= [3]_{16} \cdot 256 \qquad = [E8]_{16}$$

Big-Endian Encoding: 03 E8

Little-Endian Encoding: E8 03

⚠ Little-endian encoding of 1000 is not 8E30

Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSAsign(e, k, m)
- ▷ $R = kG$
- ▷ Define r as x-coordinate of $R = (r, R_x)$
- ▷ Compute $z = \text{hash}(m)$
- ▷ Compute $s = (z + re)/k$
- ▷ return (r, s)

ECDSAverify(P, m, r, s)
- ▷ Compute $z = \text{hash}(m)$
- ▷ Compute $u = z/s \in \mathbb{F}_p$
- ▷ Compute $v = r/s$
- ▷ Calculate testval $= u \cdot G + v \cdot P$
- ▷ If (x-coordinate of testval) $== r$ (*)

  return True

  Else

  return False

Glossary of quantities:

$e \in \mathbb{F}_p$: random number (private key)

$G \in S_{a,7}$: Generator point

$P \in S_{a,7}$: Public key, satisfies $P = e \cdot G$

$k \in \mathbb{F}_p$: random (private) nonce

$r \in \mathbb{F}_p$: public nonce (derived from private nonce)

m : message to be signed

How do we know how u, v need to be chosen?

We want: $uG + vP = R$. (this is the motivation for (*))

Since $R = kG$ and $P = eG$:

$$uG + veG = kG$$

$$\Longleftrightarrow u + ve = k$$

If $k = \frac{z + re}{s} \Rightarrow u + ve = \frac{z}{s} + \frac{r}{s} \cdot e \Rightarrow u = \frac{z}{s}$ and $v = \frac{r}{s}$ make sense.

Thus, if $u = \frac{z}{s}, v = \frac{r}{s}$, then $uG + vP = kG = R$

$\Longrightarrow$ ECDSAverify returns true if $s = \frac{z + re}{k}$ provided as input together with correct public nonce r.

Schnorr Signatures

In identity protocols above, we just provided a way for person with knowledge of private key e to show they know e without revealing e.

To create Bitcoin transactions, we also need to make sure that person with knowledge of e commits to transaction data m, and this commitment needs to be publically verifiable.

SchnorrSign(e, k, m)
- $R = kG$
- Compute $z = hash(R \| eG \| m)$
- $s = k + z \cdot e$
- return (s, R)

private key P=eG is also concatenated here to avoid that a valid signature s' for "derived public child key" P+c can be created from valid signature s for public key P.

SchnorrVerify(s, P, R, m)
- Compute sG
- Compute $z = hash(R \| P \| m)$
- Compute $testval = zP + R$
- If $sG == testval$
  - return True
- Else
  - return False

Schnorr Signature scheme as defined in BIP340 (Bitcoin Improvement Protocol 340), used with secp256k1

UNCOMPRESSED SEC FORMAT OF PUBLIC KEYS

For ECDSA: From “Standards for Efficient Cryptography (SEC)”:

Uncompressed SEC format:

1. Start with the prefix byte, which is 0x04.
2. Next, append the x coordinate in 32 bytes as a big-endian integer.
3. Next, append the y coordinate in 32 bytes as a big-endian integer.

65 bytes

Uncompressed SEC format.

Example:

```
047211a824f55b505228e4c3d5194c1fcfaa15a456abdf37f9b9d97a4040afc073dee6c8906498
4f03385237d92167c13e236446b417ab79a0fcae412ae3316b77

        - 04 - Marker
        - x coordinate - 32 bytes
        - y coordinate - 32 bytes
```

65 bytes

Compressed SEC format:

1. Start with the prefix byte. If y is even, it’s 0x02; otherwise, it’s 0x03.
2. Next, append the x coordinate in 32 bytes as a big-endian integer.

33 bytes

Compressed SEC format.

Example:

```
0349fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278a

        - 02 if y is even, 03 if odd - Marker
        - x coordinate - 32 bytes
```

33 bytes

HOW TO DERIVE Y COORDINATE FROM X COORDINATE IN COMPRESSED SEC?

Public Key $P = (x, y) \in S_{0,7} = \{(x,y) \in F_p \times F_p : y^2 = x^3 + 7\}$

Idea: Solve for y.

Recall: In Bitcoin ECDSA / Bitcoin Schnorr Scheme,
we had $p = 2^{256} - 2^{32} - 977$ (prime)

It turns out: $p \% 4 = 3$

$\Rightarrow (p+1) \% 4 = (3+1) \% 4 = 4 \% 4 = 0$

$\Rightarrow p+1$ is divisible by 4. $\Rightarrow$ $\boxed{\frac{p+1}{4} \text{ is an integer}}$

We want to find y, given z, satisfying $y^2 = z$.

$y^2 = y^2 \cdot 1 = y^2 \cdot y^{p-1} = y^{p+1}$

$z := x^3 + 7$

$y := \sqrt{z} = z^{1/2}$

what is this?

How to compute $\sqrt{\cdot}$?

Fermat's little theorem:

$y^{p-1} \% p = 1$

if y is not divisible by p.

HOW TO DERIVE Y COORDINATE FROM X COORDINATE IN COMPRESSED SEC?

We know: $p+1$ is even (since p is prime $\neq 2$)

$\Rightarrow \frac{p+1}{2}$ is integer

$$\Rightarrow y = (y^2)^{\frac{1}{2}} = (y^{p+1})^{\frac{1}{2}} = y^{\frac{p+1}{2}} = y^{2 \cdot \frac{p+1}{4}}$$

$$y^2 = z$$

$$= z^{\underbrace{\frac{p+1}{4}}}$$

is integer

$y = z^{\frac{p+1}{4}}$ gives one of the two square roots

$\Rightarrow z^{\frac{p+1}{4}}$ is computable

Assume: We know that y is even.

- If $z^{\frac{p+1}{4}}$ is even $\Rightarrow$ set $y := z^{\frac{p+1}{4}}$
- If $z^{\frac{p+1}{4}}$ is odd $\Rightarrow$ $y := p - z^{\frac{p+1}{4}}$ is even and satisfies

(since $p - z^{\frac{p+1}{4}} = -z^{\frac{p+1}{4}} \,\%\, p$) $y^2 = z$

In ECDSA: Standardized format of (r,s) output pair from signing algorithm ("Distinguished Encoding Rules")

1. Start with the 0x30 byte.
2. Encode the length of the rest of the signature (usually 0x44 or 0x45) and append.
3. Append the marker byte, 0x02.
4. Encode r as a (signed) big-endian integer, but prepend it with the 0x00 byte if r's first byte ≥ 0x80. Prepend the resulting length to r. Add this to the result.
5. Append the marker byte, 0x02.
6. Encode s as a (signed) big-endian integer, but prepend with the 0x00 byte if s's first byte ≥ 0x80. Prepend the resulting length to s. Add this to the result.

Up to 72 bytes

DER SIGNATURES

x-coordinate of R (public nonce) — signature

In ECDSA: Standardized format of (r,s) output pair from signing algorithm ("Distinguished Encoding Rules")

1. Start with the 0x30 byte.
2. Encode the length of the rest of the signature (usually 0x44 or 0x45) and append.
3. Append the marker byte, 0x02.
4. Encode r as a (signed) big-endian integer, but prepend it with the 0x00 byte if r's first byte ≥ 0x80. Prepend the resulting length to r. Add this to the result.
5. Append the marker byte, 0x02.
6. Encode s as a (signed) big-endian integer, but prepend with the 0x00 byte if s's first byte ≥ 0x80. Prepend the resulting length to s. Add this to the result.

Up to 72 bytes

DER SIGNATURES

```
3045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf213
20b0277457c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801
c31967743a9c8e10615bed

        - 30 - Marker
        - 45 - Length of sig
        - 02 - Marker for r value
        - 21 - r value lenth
        - 00ed...8f - r value
        - 02 - Marker for s value
        - 20 - s value length
        - 7a98...ed - s value
```

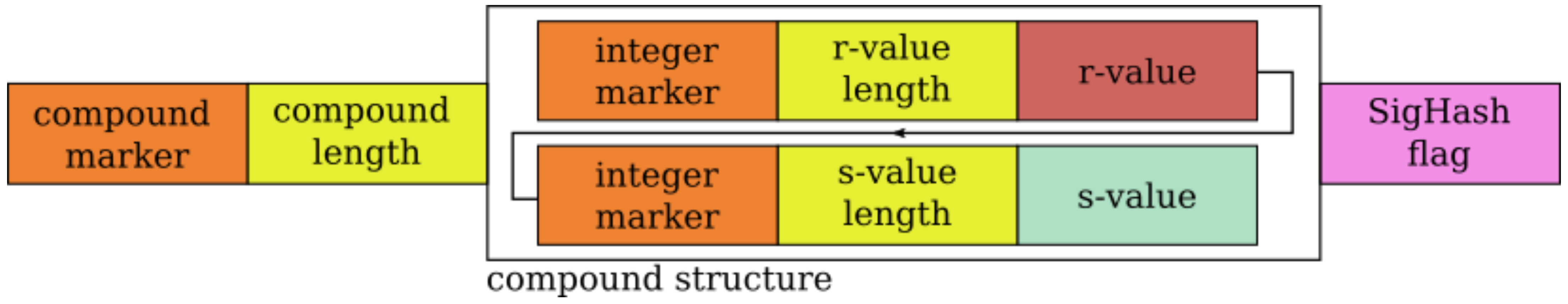
Up to 72 bytes

BITCOIN ADAPTION OF DER-ECDSA SIGNATURE FORMAT

ECDSA signatures in Bitcoin transactions are serialized in DER format with one **additional byte for a SIGHASH flag**, which describes which part of the transaction the signatures applies to.

SIGHASH flag	Value	Description
ALL	0x01	Signature applies to all inputs and outputs
NONE	0x02	Signature applies to all inputs, none of the outputs
SINGLE	0x03	Signature applies to all inputs but only the one output with the same index number as the signed input
ALL|ANYONECANPAY	0x81	Signature applies to one input and all outputs
NONE|ANYONECANPAY	0x82	Signature applies to one input, none of the outputs
SINGLE|ANYONECANPAY	0x83	Signature applies to one input and the output with the same index number

Source: https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch08_signatures.adoc

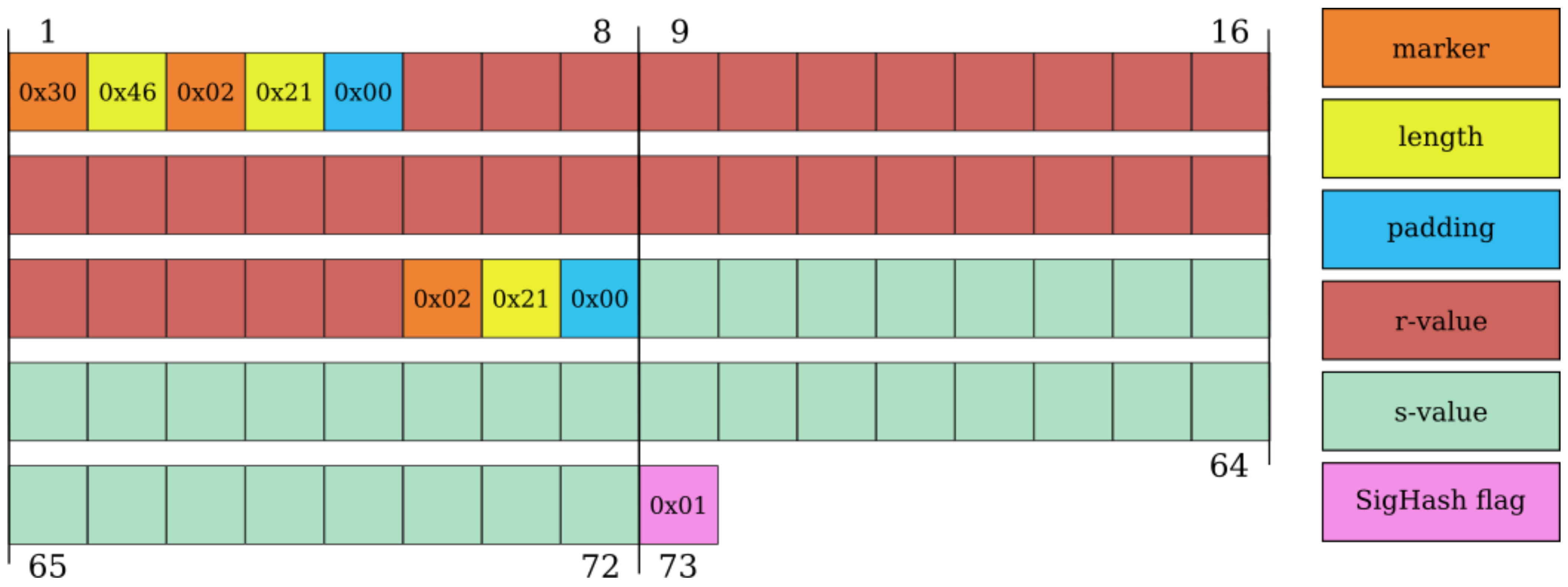


Source: https://b10c.me/blog/006-evolution-of-the-bitcoin-signature-length/

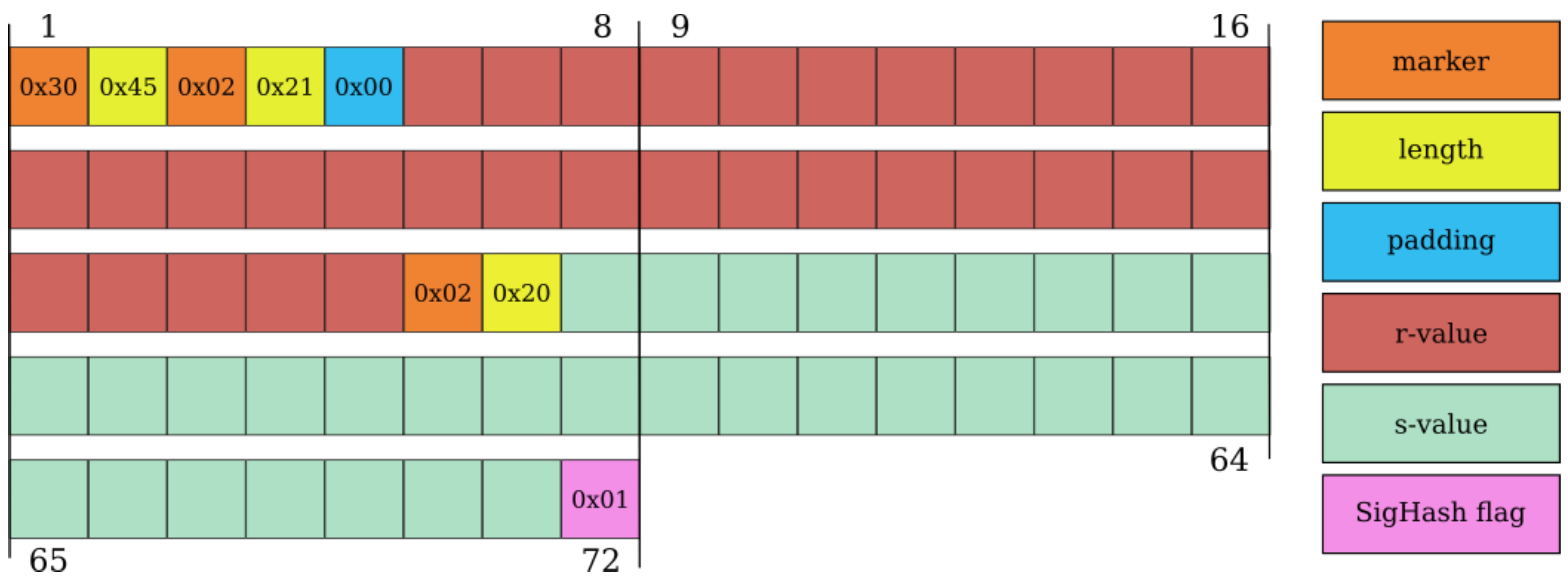
Length of DER-encoded Bitcoin ECDSA signatures: **Up to 73 bytes**

BITCOIN ADAPTION OF DER-ECDSA SIGNATURE FORMAT

[Statistics about ECDSA signatures in Bitcoin transactions]

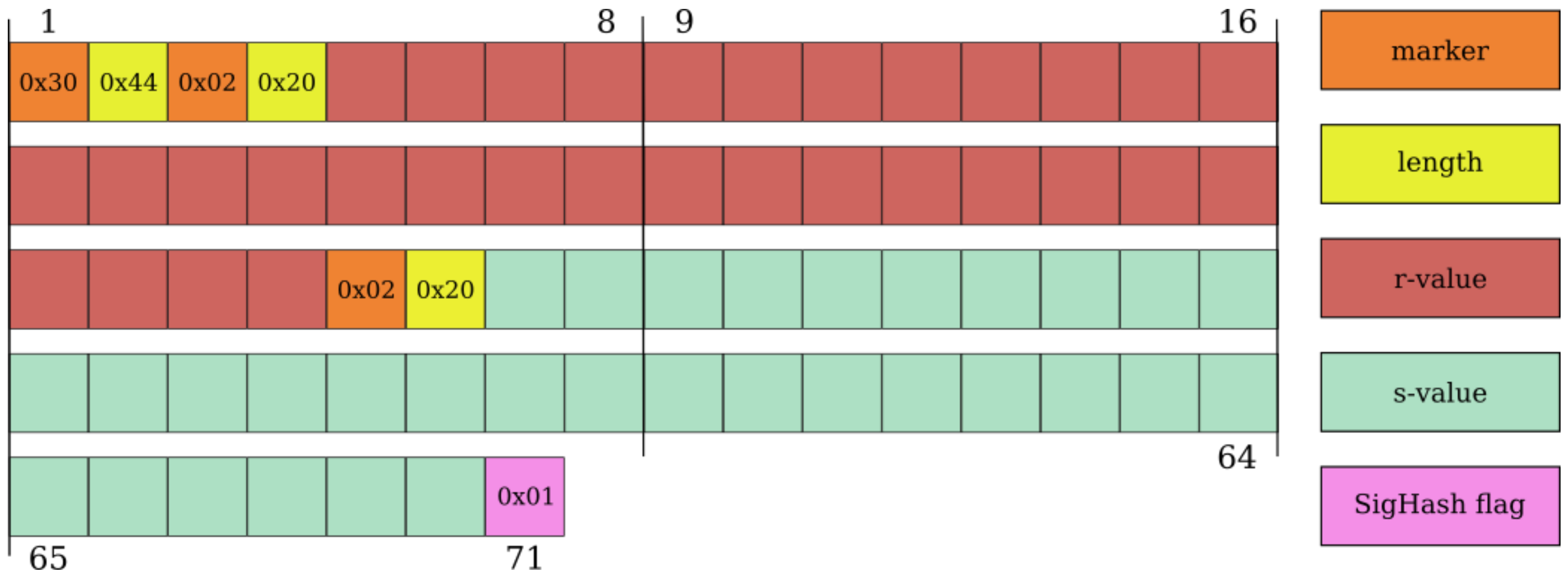


73-byte “high-r” and “high-s” Bitcoin ECDSA signature



72-byte “high-r” and “low-s” Bitcoin ECDSA signature

Up to 73 bytes

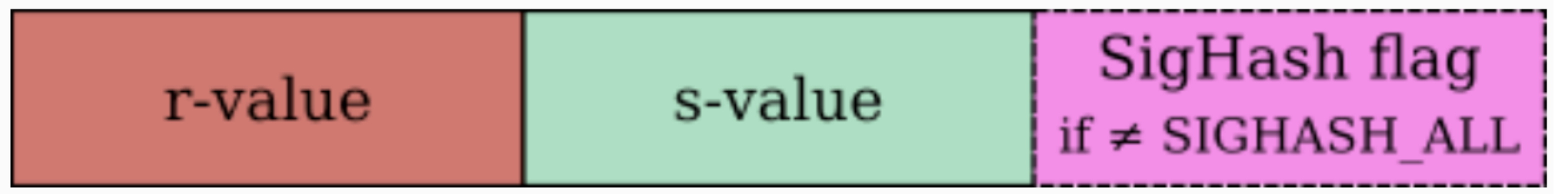


71-byte “low-r” and “low-s” Bitcoin ECDSA signature

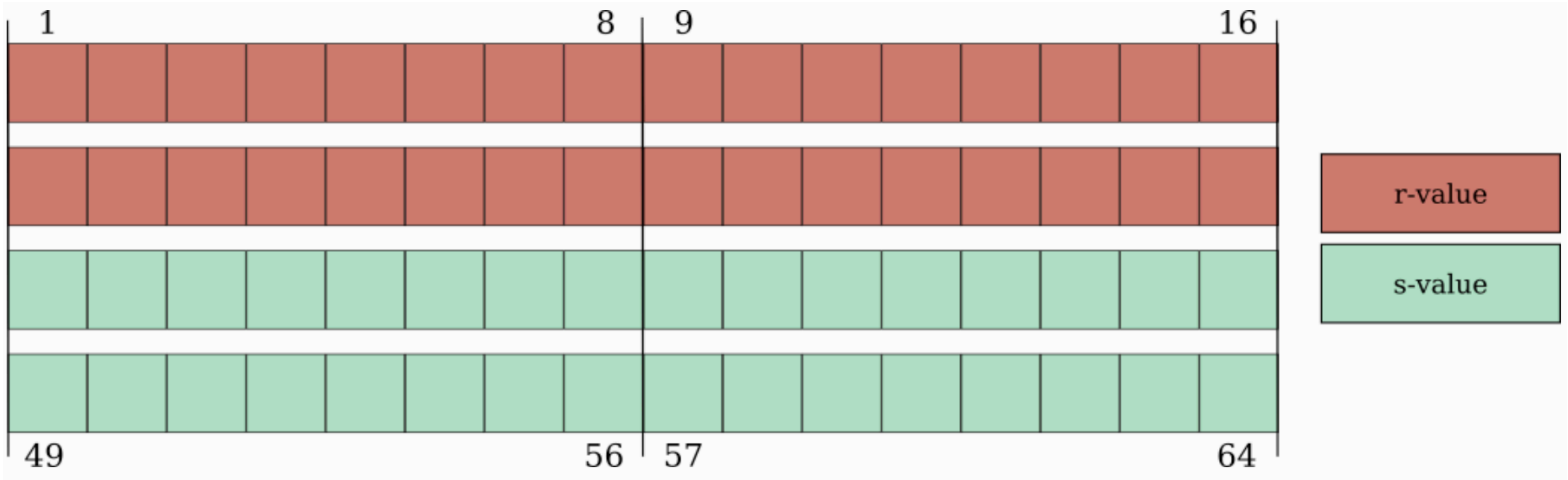
Source: [https://b10c.me/blog/006-evolution-of-the-bitcoin-signature-length/]

SERIALIZATION OF SCHNORR SIGNATURES

To ensure cheaper serialization, [BIP 340 implementation of Schnorr signatures]() makes sure that the elliptic curve point $R = k \cdot G$, where k is private nonce, has always an even y-coordinate => It is sufficient to store only x-coordinate r of R.



Serialization format of Schnorr signatures in Bitcoin (BIP-340)



A 64-byte SIGHASH_ALL Bitcoin Schnorr signature

Up to 65 bytes

Compared to serialized ECDSA signatures, Schnorr signatures are 6-9 bytes shorter.

Source: https://b10c.me/blog/006-evolution-of-the-bitcoin-signature-length/