Bitcoin
Elliptic Curve Digital Signature Algorithm (ECDSA)

Bassam El Khoury Seguias
BTC: 3FcVvBZwTUkUrcqJd16RcjR42qT2tDWHWn

September 8, 2018

1 Introduction

Simply stated, a bitcoin transaction is a transfer of **spending control** between different parties over a pre-specified amount of satoshis. A satoshi is the smallest fraction of a bitcoin and is equivalent to BTC 10^{-8}. In order to successfuly complete said transfer, the sender must demonstrate that she is the rightful owner of the satoshis she wishes to spend. Such a proof is imperative as it allows the different nodes on the network to reach an agreement regarding the validity of the transaction and as a result, facilitate its inclusion in the blockchain.

At the time of writing, bitcoin's proof of ownership is encapsulated in a particular type of digital signature known as the Elliptic Curve Digital Signature Algorithm (**ECDSA**). It is a variant of the Digital Signature Algorithm (**DSA**) that relies on Elliptic Curve Cryptography (**ECC**).

In the first section, we introduce the DSA scheme, prove its **correctness**, and discuss some of its **security** properties. In particular, we point out that as of the time of writing, and despite its prevalence in various cryptographic settings, we do not know of any valid security proof of DSA in the random oracle (RO) model. However, we highlight that slight variations of it can be proven to be secure.

In the second section, we introduce the ECDSA scheme and prove its **correctness**. Later on, we present a python-based implementation to further elucidate its building blocks. We also describe how an ECDSA signature gets typically encoded within a bitcoin transaction. Finally, we highlight some of the scheme's potential shortcomings including the absence to-date of a **security** proof in the RO model, its susceptibility to being **malleable**, and its non-linear design that hinders an efficient implementation of **multisignature** transactions.

In order to better understand the material contained herein, we recommend that the reader familiarizes himself with the necessary prerequisites fleshed out in the following three posts:

1. *Groups and Finite Fields*
2. *Elliptic Curve Groups*
3. *Digital Signatures and other Prerequisites*

2 Digital Signature Algorithm (DSA)

The invention of the Digital Signature Algorithm (DSA) is attributed to David W. Kravitz [9] who used to work for the National Security Agency (NSA). The legitimacy of this invention has been contested by Claus Schnorr (the inventor of the Schnorr signature scheme), who asserted that DSA is covered in another patent of his [12]. Readers interested in the claims and counterclaims surrounding the origin of DSA can refer to e.g., [2].

The DSA scheme is built on the finite fields $\mathbb{Z}_q$ and $\mathbb{Z}_p$, where q and p are two large prime numbers of respective bit-lengths N and L, and such that q is a divisor of $p-1$. We can think of N as the scheme's security parameter. Let g be an element of order q in the multiplicative cyclic subgroup $(\mathbb{Z}_p^*,\ \otimes_p)$. One way of finding such a g consists in letting $g \equiv a^{(p-1)/q} \pmod{p}$, for arbitrary $a \in (\mathbb{Z}_p^*,\ \otimes_p)$ such that $a^{(p-1)/q} \not\equiv 1 \pmod{p}$. To see why this construction works, note that:

- By Lagrange's theorem, we know that $\forall a \in (\mathbb{Z}_p^*,\ \otimes_p)$, order$(a)$ divides $p-1$. Let order$(a) \equiv m = \frac{p-1}{d}$ for some integer $d \leq (p-1)$.
- We then have $g^q \equiv a^{(p-1)} \equiv (a^m)^d \equiv 1 \pmod{p}$. This guarantees that order(g) must be a divisor of q. But since q is prime, it must be that order(g) is either equal to q or equal to 1.
- If order(g) were equal to 1, we would have $a^{(p-1)/q} \equiv g = g^1 \equiv 1 \pmod{p}$. But by definition, we required that $a^{(p-1)/q} \not\equiv 1 \pmod{p}$. As a result, it must be that order$(g) = q$.

Similarly to other digital signature schemes, we define DSA as a set of three algorithms:

- **The DSA key generation algorithm** $\mathcal{G}$. On input 1^N, it produces a pair (x, y) of matching private and public keys where x is a randomly chosen element in $\mathbb{Z}_q^*$ and $y \equiv g^x \pmod{p}$. The algorithm is modeled as a PPT Turing machine.
- **The DSA signing algorithm** Σ. Suppose a user with private and public key pair (x, y) decides to sign a message m. Let $\mathcal{H}$ be an appropriate hashing function (e.g., SHA-256) and let

$$t : \{0,1\}^* \longrightarrow \cup_{l=1}^{N} \{0,1\}^l$$

be the truncation function mapping strings of arbitrary length to strings of length at most N and such that:

- It acts as the identity map if the input string is of length at most N
- It outputs the N least significant bits of its argument otherwise

Σ proceeds as follows:

1. Let $z = t \circ \mathcal{H}(m)$
2. Select a random element $k \in \mathbb{Z}_q^*$. We point out that it is crucial to select a different random k for each signature instance. Otherwise, an adversary could use two signatures sharing the same k to deduce the common value and subsequently break the signer's private key as we explain later on.
3. Set $r \equiv (g^k \pmod p) \pmod q$. If $r = 0$, go back to the previous step and choose another random k.
4. Compute $s = k^{-1} \times (z + (x \times r)) \pmod q$. Here, k^{-1} is the mutiplicative inverse of k in modulo q arithmetic and we have previously seen in the prerequisite sections how to compute it efficiently using the Extended Euclidean Algorithm. If $s = 0$, go back to the first step and choose another random k.

Σ finally outputs a signature $\sigma(m) \equiv (r, s)$. The algorithm is modeled as a PPT Turing machine.

- **The DSA verification algorithm $\mathcal{V}$.** Given a signature σ, a message m, and the public key y of the presumable signer, $\mathcal{V}$ verifies the validity of $\sigma(m)$ by checking the following:
  - If $r \notin \mathbb{Z}_q^*$ or $s \notin \mathbb{Z}_q^*$, $\mathcal{V}$ outputs *False*.
  - Otherwise:
    - Compute $u = (t \circ \mathcal{H}(m)) \times s^{-1} \pmod q$ and $v = r \times s^{-1} \pmod q$ where s^{-1} is the multiplicative inverse of s in modulo q arithmetic.
    - Compute $w = (g^u \times y^v \pmod p) \pmod q$
    - If $w = r$, then $\mathcal{V}$ outputs *True*. Otherwise, it outputs *False*.

  $\mathcal{V}$ is a deterministic algorithm as opposed to probabilistic.

Correctness of DSA The DSA scheme satisifies the correctness property. In other words, any signature generated by Σ will cause the verification algorithm to output *True*. To see why, let $\sigma(m) \equiv (r, s)$ be an appropriate signature on message m. First note the following chain of implications:

$$s = k^{-1} \times (z + (x \times r)) \pmod q \Rightarrow$$

$$k \times s = (z + (x \times r)) \pmod q \Rightarrow$$

$$k = (z + (x \times r)) \times s^{-1} \pmod q \Rightarrow$$

$$k = [(z \times s^{-1}) \pmod q + (x \times r \times s^{-1}) \pmod q] \pmod q$$

Recalling that order$(g) = q$, and noting that for some appropriate integer α we have

$$[(z \times s^{-1}) \pmod q \; + \; (x \times r \times s^{-1}) \pmod q] = \alpha q + k, \text{ we get:}$$

$$g^k = g^{\alpha q} \times g^k \pmod p = g^{(z \times s^{-1}) \pmod q} \times g^{(x \times r \times s^{-1}) \pmod q} \pmod p$$

$$= \; g^u \times g^{\{x \; \times [(r \times s^{-1}) \pmod q)]\} \pmod q} \pmod p$$

Similarly, for some appropriate integer α', we can write:

$$x \; \times [(r \times s^{-1}) \pmod q)] = \alpha' q + \{x \; \times [(r \times s^{-1}) \pmod q)]\} \pmod q$$

Using one more time the fact that order$(g) = q$, we get:

$$g^k = g^u \times (g^x)^{(r \times s^{-1}) \pmod q} \pmod p = g^u \times y^v \pmod p$$

Upon verification, algorithm $\mathcal{V}$ computes $w = (g^u \times y^v \pmod p) \pmod q$ and concludes that $w = (g^k \pmod p) \pmod q = r$, based on the previous equality. This result shows that the output of Σ satisfies the verification algorithm $\mathcal{V}$ hence demonstrating the correctness of DSA.

Security of DSA: The importance of the randomness of the parameter k. A necessary condition for the DSA scheme to be secure is for the parameter k to be used once per each signature instance. Indeed, if this were not the case, one would be able to derive the private key x of the signer. To see why, suppose that $\sigma(m_1) \equiv (r_1, s_1)$ and $\sigma(m_2) \equiv (r_2, s_2)$ are two signatures generated by the same signer with private key x and such that $k_1 = k_2 = k$. We write:

$$s_1 - s_2 \; = \; k^{-1} \times [z_1 + (x \times r_1) - z_2 - (x \times r_2)] \pmod q$$

By design of Σ, we have $r_1 \equiv (g^k \pmod p) \pmod q = r_2$. We then get:

$$s_1 - s_2 \; = \; k^{-1} \times (z_1 - z_2) \pmod q$$

This allows us to solve for the parameter k as follows:

$$k = (z_1 - z_2) \times (s_1 - s_2)^{-1} \pmod q$$

Finally, note that Σ mandates that $s \; = \; k^{-1} \times (z + (x \times r)) \pmod q$. This implies that $x \; = \; [(k \times s) - z] \times r^{-1} \pmod q$. Consequently, one can retrieve x by using either signatures (s_1, r_1) or (s_2, r_2), the hash of the corresponding message $\mathcal{H}(m_1)$ or $\mathcal{H}(m_2)$, and the common value k.

Security of DSA: A note on existential unforgeability. A security proof for a digital signature scheme is essentially a proof of resilience against existential forgeries in the adaptive chosen-message setting (EFACM). The rather odd observation is that despite the widespread adoption of DSA, there is no known proof of its security in the

RO model. Typically, such proofs rely on a reduction technique that transforms a hypothetically successful forgery attack into a solution to a computational problem believed to be hard (e.g., finding discrete logarithms over certain finite groups).

Before proceeding further, we recommend that the reader familiarizes himself with the content of the following two posts for a better understanding of the logic outlined in this section:

- *Digital Signature and other Prerequisites* to learn more about digital signatures, forgeries, and security proofs.
- *Pointcheval & Stern's Generic Signature Scheme* to see the reduction technique in action as it applies to e.g., the Schnorr signature scheme.

The belief that a DSA security proof may be difficult to construct rests on our inability to date to successfully leverage the reduction model (RM) in that respect. However, it is important to note that the absence of a proof at the time of writing does not imply that a proof does not exist. In what follows, we attempt to argue why a DSA security proof based on RM may be difficult to devise. To do so, we will need to revisit the foundational steps of the model as outlined in the aforementioned posts.

Recall that RM applies a reductio ad absurdum argument that starts by assuming that the signature scheme is not secure i.e., there exists a PPT adversary $\mathcal{A}$ such that:

$$P_{\omega,r,\mathcal{H}}[\mathcal{A}(\omega)^{\mathcal{H},\Sigma^{\mathcal{H}}(r)} \text{ succeeds in } EFACM] = \epsilon(N)$$

where ω is $\mathcal{A}$'s random tape, r is the random tape of DSA's signing algorithm Σ (not to be confused with the r that appears in ECDSA's signature), $\mathcal{H}$ is the random oracle, N is DSA's security parameter and ϵ a quantity non-negligible in N.

Subsequently, the model applies a series of steps that culminate in the extraction of a solution to a problem thought to be computationally hard e.g., finding the private key x associated with a given public key y. In DSA, one way of solving for the key consists in devising two distinct valid signatures $\sigma(m_1) \equiv (r_1, s_1)$ and $\sigma(m_2) \equiv (r_2, s_2)$, leading to a linear equation in x. Conditions $\mathbf{C}_1$ and $\mathbf{C}_2$ below are jointly sufficient for this to be possible:

$$\mathbf{C}_1) \;\; g^{u_1} \times y^{v_1} \; = \; g^{u_2} \times y^{v_2} \pmod p$$

$$\mathbf{C}_2) \;\; v_1 \neq v_2$$

To see why, substitute y with g^x and write $\mathbf{C}_1$ as $g^{u_1+xv_1} = g^{u_2+xv_2} \pmod p$. Since order$(g) = q$, this implies that $u_1 + xv_1 = u_2 + xv_2 \pmod q$. $\mathbf{C}_2$ would then allow us to solve for $x = (u_1 - u_2) \times (v_2 - v_1)^{-1} \pmod q$.

In what follows, we derive necessary conditions for $\mathbf{C}_1$ and $\mathbf{C}_2$ to hold. We then argue why applying RM to the DSA scheme does not imply with certainty that these necessary conditions actually hold. This means that we cannot imply with certainty

that $\mathbf{C}_1$ and $\mathbf{C}_2$ hold, leading us to conclude that solving for x may be difficult after all. We reiterate that we are not arguing that a security proof for DSA is not possible, but rather that such a proof may be difficult to achieve using the reduction technique.

First, since both signatures are valid, the verification equations guarantee that:

$$(g^{u_i} \times y^{v_i} \pmod p) \pmod q \; \equiv w_i \; = \; r_i \equiv (g^{k_i} \pmod p) \pmod q \text{ , for } i = 1, 2$$

As a result:

$$\mathbf{C}_1 \Rightarrow \; r_1 = r_2 \iff (g^{k_1} \pmod p) \pmod q = (g^{k_2} \pmod p) \pmod q.$$

Consequently, k_1 and k_2 must exhibit a certain relationship for the first condition to hold. With overwhelming probabilty, two randomly chosen parameters k_1 and k_2 will not satisfy this equality.

Since $\mathbf{C}_1 \Rightarrow r_1 = r_2$, and since $v_i = r_i \times s_i^{-1} \pmod q$, for $i = 1, 2$, we can write:

$$\mathbf{C}_1 \cap \; \mathbf{C}_2 \Rightarrow (r_1 = r_2) \; \cap \; (s_1 \neq s_2)$$

We also know that $s_i = k^{-1} \times (z_i + (x \times r_i)) \pmod q$ for $i = 1, 2$. This yields:

$$(r_1 = r_2) \; \cap \; (s_1 \neq s_2) \; \Rightarrow \; (k_1 \neq k_2) \; \cup \; (\mathcal{H}(m_1) \neq \mathcal{H}(m_2))$$

The takeaway is that to be able to effectively use the reduction technique to solve for x in the case of DSA, one will **possibly** need to ensure that valid signatures $\sigma(m_1)$ and $\sigma(m_2)$ satisfy the following at a minimum:

1. $(g^{k_1} \pmod p) \pmod q = (g^{k_2} \pmod p) \pmod q$, and
2. $\mathcal{H}(m_1) \neq \mathcal{H}(m_2)$ or $k_1 \neq k_2$.

We deliberatly used the term "possibly" and the justification is two-fold:

a In the above, we limit ourselves to a particular type of computationally difficult problem, namely solving a discrete logarithm over a multiplicative cyclic subgroup. Nothing prevents anyone from considering other hard problems for which $\mathbf{C}_1$ and $\mathbf{C}_2$ could become obsolete.

b Conditions $\mathbf{C}_1$ and $\mathbf{C}_2$ were sufficient to solve for x but not undoubtedly necessary.

In what follows, we apply the remaining steps of RM and for argument's sake, we assume that steps two and three hold. In other words, we assume that we are able to:

- Build an adequate simulator $\mathcal{S}(r')$ where r' is the simulator's random tape.
- Show that the probability of faulty collisions in this simulated environment is negligible.

In the fourth step, one would show that an adversary that can forge a signature, is also capable with non-negligible probability of creating a second forgery distinct from the

first one. Most importantly, the two forgery attacks would behave in a similar way up to a certain well-defined event. Formally, one would show that the following quantity is non-negligible in N:

$$P_{\mathcal{H}}[\mathcal{A}(\omega^*)^{\mathcal{H},\mathcal{S}(r'^*)}\ succeeds\ in\ EFACM\ \cap\ (\rho_\beta \neq \rho^*_\beta) \mid (\omega^*, r'^*, \mathcal{H}^*)\ is\ a\ succesfull\ first\ forgery,\ and\ (\rho_i = \rho^*_i)\ for\ i \in \{1, ...\beta - 1\}]$$

When producing its first forgery $(\omega^*, r'^*, \mathcal{H}^*)$, the adversary $\mathcal{A}$ is assumed to make a number n of queries to random oracle $\mathcal{H}$. We denote the ith query as q_i and its corresponding random oracle reply as ρ_i (i.e., $\rho_i \equiv \mathcal{H}(q_i), i = 1, ..., n$). The second forgery is created through a process known as an "oracle replay attack" and consists of:

- Using the same random tape r'^* of the simulator $\mathcal{S}$ used in the first attack.
- Using the same random tape ω^* of the adversary $\mathcal{A}$ used in the first attack.
- Ensuring that the replies of the random oracle $\mathcal{H}$ queried in the first and second attacks match up to the $(\beta - 1)^{th}$ query $(2 \leq \beta \leq n)$, after which the replies could start to diverge.

The standard forking lemma applied in RM subsequently leads to the following result: Given a successful forgery tuple $(\omega^*, r'^*, \mathcal{H}^*)$, we can find with non-negligible probability another successful forgery tuple $(\omega^*, r'^*, \mathcal{H}^\sim)$ such that

$$(\rho^\sim_1 = \rho^*_1), ..., (\rho^\sim_{\beta-1} = \rho^*_{\beta-1}), \text{ but } (\rho^\sim_\beta \neq \rho^*_\beta).$$

We let $(\omega^*, r'^*, \mathcal{H}^*)$ correspond to $\sigma_{forge}(m_1) \equiv (r_1, s_1)$, and $(\omega^*, r'^*, \mathcal{H}^\sim)$ correspond to $\sigma_{forge}(m_2) \equiv (r_2, s_2)$. At this stage, we highlight two important observations:

1. Adversary $\mathcal{A}$ is bound to query $\mathcal{H}$ on input m_1 at some point during the first forgery experiment. To see why, note that any successful forgery must pass the verification test. If $\mathcal{A}$ never queried $\mathcal{H}$ on input m_1, the probability of it generating a successful forgery would be upperbounded by the probability that $\mathcal{H}(m_1)$ (i.e., the value that RO returns to $\mathcal{V}$ on query m_1) satisfies the verification condition, namely:

   $$r_1 = (g^{u_1} \times y^{v_1} \pmod p) \pmod q = g^{(z_1 + x \times r_1) \times (s_1)^{-1}} \pmod p) \pmod q$$

   Since z_1 could be any value in $\mathbb{Z}^*_q$, and since order(g) is a prime number equal to q, the probability of such an event is on the order of $\frac{1}{q-1}$ which is negligible in N.

2. Since the two forgery experiments have the same random tapes ω^* and r'^*, and since $\mathcal{H}^*$ and $\mathcal{H}^\sim$ behave the same way on the first $\beta - 1$ queries, we can be confident that the first β queries sent to the RO in the two experiments are identical. In particular the two β^{th} queries are the same.

In light of the above observations, if we let β be the index of the query corresponding to m_1, we can ensure that $m_1 = m_2$ while $\mathcal{H}(m_1) \neq \mathcal{H}(m_2)$. This will satisfy part of the necessary conditions that we discussed earlier for $\mathbf{C}_1 \cap\ \mathbf{C}_2$ to hold. The issue however, is with enforcing the remaining part of the necessary conditions, namely that k_1 and k_2

be appropriately linked. The reason this is difficult to enforce is because there is no guarantee that k gets selected by $\mathcal{A}$ before m is queried to $\mathcal{H}$. k could very well be specified after the βth query causing the two signatures to have associated parameters k_1 and k_2 without any particular binding relationship.

Due to this limitation, one cannot conclude with certainty that DSA is necessarily secure in the RO model. However, slight variants of the DSA scheme can be shown to be secure. We mention below two such variants due to Brickell [5] and to Pointecheval and Vaudenay [11]:

1. The **Brickell** variant replaces the $\pmod q$ operation that appears in the calculation of r in DSA's Σ algorithm and in the calculation of w in DSA's $\mathcal{V}$ algorithm by a random oracle $\mathcal{H}_2$. In other words, r becomes $\mathcal{H}_2(g^k \pmod p)$ and w becomes $\mathcal{H}_2((g^u \times y^v) \pmod p)$. We refer the reader to [5] for a proof of its security.
2. The **Pointecheval-Vaudenay** variant takes the hash of m and r combined instead of that of m alone (i.e., $\mathcal{H}(m, r)$ instead of $\mathcal{H}(m)$). This construct seems to be the most natural, especially in light of the previous discussion about the difficulty of building a security proof for DSA. One can only wonder why the NSA did not adopt this formulation as part of its original scheme. We refer the reader to [11] for a proof of its security.

Aside from these variants, Fersch et al. [8] devised a security proof for the unmodified version of DSA by introducing an extra modeling constraint. This constraint, also known as the **bijective random oracle**, applies to the **conversion function**:

$$f : \mathbb{N} \longrightarrow \mathbb{Z}_q^*$$

$$e \longrightarrow e \pmod p \pmod q$$

The conversion function is none else than the one that DSA's signature algorithm Σ uses to calculate r with group element g^k as input. The constraint that Fersch et al. impose consists of representing f as a composition of three functions $\psi \circ \Pi \circ \phi$, such that Π is a bijection and such that both Π and Π^{-1} are modeled as random oracles. We will not go over the details of their proof, but the interested reader can refer to [8].

3 The ECDSA scheme

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a variant of DSA that uses Elliptic Curve Cryptography (ECC), a topic that we previously introduced in the post entitled *Elliptic Curve Groups*. For a given public key length, ECC bestows on ECDSA a significant security advantage over its DSA counterpart. This advantage is a consequence of the observation that the security of cryptographic primitives built on the presumed hardness of the Elliptic Curve Discrete Logarithm Problem (ECDLP) surpasses that of those built on the presumed hardness of the Discrete Logarithm Problem (DLP) on multiplicative cyclic subgroups.

To put this comparative advantage into perspective, we point out that the difficulty of solving ECDLP with 160-bit long public keys is comparable to that of solving DLP on a multiplicative cyclic subgroup with 1024-bit long public keys [3]. In this context, the notion of difficulty refers to the expected amount of time needed to break the discrete logarithm problem.

Being an ECC primitive, ECDSA requires signers and verifiers to agree on the parameters of the elliptic curve to be used. For bitcoin, the curve is **secp256k1** whose defining parameters were previously introduced in the *Elliptic Curve Groups* post. We relist them below for ease of reference:

- The elliptic curve group associated with secp256k1 is $(E(\mathbb{F}_p), \oplus_p)$, where

$$E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p^2 \mid y^2 \equiv x^3 + 7 \pmod{p}\} \ \cup \ \{\mathcal{O}\}$$

  - p is the prime number equal to $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$.
  - $\oplus_p$ is the operation denoting elliptic curve point addition.
  - $\mathcal{O}$ is the point at infinity.

- The chosen base point G of $(E(\mathbb{F}_p), \oplus_p)$, has decimal coordinates given by

$$x_G \equiv 55066263022277343669578718895168534326250603453777594175500187360389116729240 \pmod{p}$$

$$y_G \equiv 32670510020758816978083085130507043184471273380659243275938904335757337482424 \pmod{p}$$

  Bitcoin's public-key cryptography is hence conducted on the subgroup $(\{G\}, \oplus_p)$.

- Moreover, the order of G is chosen to be a prime number equal to

$$n = 115792089237316195423570985008687907852837564279074904382605163141518161494337 \pmod{p}$$

- It turns out that the cardinality of $(E(\mathbb{F}_p), \oplus_p)$ is equal to n which is a prime number. As a result, it is a cyclic group and any of its elements could serve as a generator (refer to *Groups and Finite Fields* for an introduction to cyclic groups).

We let $L_n \equiv \lfloor log_2(n) + 1 \rfloor$ denote the security parameter associated with ECDSA. In what follows we define this signature scheme as a set of three algorithms:

- **The ECDSA key generation algorithm** $\mathcal{G}$. On input $1^{(L_n)}$, it produces a pair (d, H) of matching private and public keys where d is a random element in $\mathbb{Z}_n^*$ and H is the elliptic curve point given by $d \otimes_p G \ \equiv \ G \ \oplus_p \ ... \ \oplus_p \ G$ (d times). The algorithm is modeled as a PPT Turing machine.

- **The ECDSA signing algorithm** Σ. Suppose a user with private and public key pair (d, H) decides to sign a message m. Moreover, let $\mathcal{H}$ be an appropriate hashing function (e.g., SHA-256), and let

$$t : \{0,1\}^* \longrightarrow \cup_{l=1}^{L_n} \{0,1\}^l$$

be the truncation function mapping strings of arbitrary length to strings of length at most L_n and such that:

  - It acts as the identity map if the input string is of length at most L_n
  - It outputs the L_n least significant bits of its argument otherwise

Σ proceeds as follows:

  1. Let $z = t \circ \mathcal{H}(m)$
  2. Select a random element $k \in \mathbb{Z}_n^*$. As was the case for DSA, it is crucial for ECDSA to select a different random k for each signature instance. A similar proof to the DSA case demonstrates that failure to do so would jeopardize the private key.
  3. Compute the elliptic curve point

$$P \equiv (x_p, y_p) = k \otimes_p G \ \equiv \ G \ \oplus_p \ ... \ \oplus_p \ G \ (k \text{ times})$$

  4. Set $r \equiv x_p \pmod{n}$. If $r = 0$, go back to step 2 and choose another random k.
  5. Compute $s = k^{-1} \times (z + (d \times r)) \pmod{n}$ where k^{-1} is the multiplicative inverse of k in modulo n arithmetic. Note that as shown in the post on *Groups and Finite Fields*, if n were not prime this inverse could not be guaranteed to exist for arbitrary k. If $s = 0$, go back to step 2 and choose another random k.

Σ finally outputs a signature $\sigma(m) \equiv (r, s)$. The algorithm is modeled as a PPT Turing machine.

- **The ECDSA verification algorithm** $\mathcal{V}$. Given a signature σ, a message m, and the public key H of the presumable signer, $\mathcal{V}$ verifies the validity of $\sigma(m)$ by checking the following:

  - If H does not satisfy the elliptic curve equation or if $H = \mathcal{O}$ (i.e., the identity element of the elliptic curve group), $\mathcal{V}$ outputs *False.*
  - If r or $s \notin \mathbb{Z}_n^*$, $\mathcal{V}$ outputs *False.*
  - Otherwise:
    - Compute $u = [t \circ \mathcal{H}(m)] \times s^{-1} \pmod{n}$ and $v = r \times s^{-1} \pmod{n}$ where s^{-1} is the multiplicative inverse of s in modulo n arithmetic.
    - Compute $W \equiv (x_w, y_w) = (u \ \otimes_p \ G) \ \oplus_p \ (v \ \otimes_p \ H)$
    - If $r = x_w \pmod{n}$ then $\mathcal{V}$ outputs *True.* Otherwise, it outputs *False.*

$\mathcal{V}$ is a deterministic algorithm as opposed to probabilistic.

Correctness of ECDSA The ECDSA scheme satisifies the correctness property. In other words, any signature generated by Σ will cause the verification algorithm to output *True*. To prove it, we follow a similar logic to the one used to prove DSA's correctness. More specifically, let $\sigma(m) \equiv (r, s)$ be a signature on message m and note that:

$$s = k^{-1} \times (z + (d \times r)) \pmod n \Rightarrow$$

$$k \times s = (z + (d \times r)) \pmod n \Rightarrow$$

$$k = (z + (d \times r)) \times s^{-1} \pmod n \Rightarrow$$

The verification algorithm $\mathcal{V}$ will then compute

$$W = (u \otimes_p G) \oplus_p (v \otimes_p H) \Rightarrow$$

$$W = (u \otimes_p G) \oplus_p [(d \times v \pmod n)) \otimes_p G] \Rightarrow$$

$$W = \{[(t \circ \mathcal{H}(m)) \times s^{-1} \pmod n] \otimes_p G\} \oplus_p \{(d \times r \times s^{-1} \pmod n)) \otimes_p G\} \Rightarrow$$

$$W = [(z + (d \times r)) \times s^{-1} \pmod n] \otimes_p G = k \otimes_p G = P$$

The previous equality allows us to conclude that:

$$r \equiv x_p \pmod n = x_w \pmod n,$$

hence validating $\sigma(m)$ and establishing ECDSA's correctness.

Illustrative implementation in python. In what follows, we show how the ECDSA signature scheme could be implemented in python. Note that it is always recommended to rely on existing and well-tested implementations. The one below is for educational purposes and we built it from scratch with the sole intention of illustrating the process.

ECDSA relies on elliptic curve point addition and scalar multiplication. We include below five python methods, the first three of which feed into **mul_scalar** that perfoms elliptic-curve point multiplication. The last method verifies whether a point belongs to a pre-specified elliptic curve or not. The first two methods were sourced from [7].

1. **extended_euclidean_algorithm(a, b)**: it takes two integers a and b and returns a three-tuple consisting of $\gcd(a, b)$ and the bézout coefficients x and y that satisfy $ax + by = \gcd(a, b)$ (refer to *Groups and Finite Fields*):

```
def extended_euclidean_algorithm(a, b):
    s, old_s = 0, 1
    t, old_t = 1, 0
    r, old_r = b, a

    while r != 0:
        quotient = old_r // r
        old_r, r = r, old_r - quotient * r
        old_s, s = s, old_s - quotient * s
        old_t, t = t, old_t - quotient * t

    return old_r, old_s, old_t
```

2. **inverse_of(n,p)**: it computes the inverse of n mod p by relying on the **extended_euclidean_algorithm** method (refer to *Groups and Finite Fields*):

```
def inverse_of(n, p):
    gcd, x, y = extended_euclidean_algorithm(n, p)
    assert (n * x + p * y) % p == gcd

    if gcd != 1:
        # Either n is 0, or p is not a prime number.
        raise ValueError(
            '{} has no multiplicative inverse '
            'modulo {}'.format(n, p))
    else:
        return x % p
```

3. **add_points(A, B, p, p1, p2)**: it adds two points $p1$ and $p2$ on the short Weierstrass form elliptic curve whose equation is

$$E : y^2 \equiv x^3 + Ax + B \pmod{p}$$

```
def add_points(A, B, p, p1, p2):
    if (p1 == "0"): return p2        # "0" denotes the identity element of the group
    elif (p2 == "0"): return p1

    else:
        x_p1, y_p1 = p1
        x_p2, y_p2 = p2

        if (not ((x_p1 - x_p2) % p) and ((y_p1 - y_p2) % p)): return "0"
        elif (not ((x_p1 - x_p2) % p) and (not ((y_p1 - y_p2) % p)) and (not (y_p1 % p))):
            return "0"
        elif (not ((x_p1 - x_p2) % p) and (not ((y_p1 - y_p2) % p)) and (y_p1 % p)):
            c = ((3*x_p1**2 + A) * inverse_of(2*y_p1, p)) % p
            d = (y_p1 - c*x_p1) % p
            x_p12 = (c**2 - 2*x_p1) % p
            return (x_p12, (-c*x_p12 - d) % p)
        elif ((x_p1 - x_p2) % p):
            c = ((y_p2 - y_p1) * inverse_of(x_p2 - x_p1, p)) % p
            d = ((x_p2 * y_p1 - x_p1 * y_p2) * inverse_of(x_p2 - x_p1, p)) % p
            x_p12 = (c**2 - x_p1 - x_p2) % p
            return (x_p12, (-c*x_p12 - d) % p)
```

4. **mul_scalar(A, B, p, p1, m)**: it multiplies a scalar m by a point $p1$ on the short Weierstrass form elliptic curve whose equation is

$$E : y^2 = x^3 + Ax + B \pmod p$$

It implementats the **double-and-add** algorithm previously introduced in the *Elliptic Curve Groups* post and it relies on the **add_points** method

```
def mul_scalar(A, B, p, p1, m):
    output = "0"
    while m > 0:
        # If parity is odd, then add a single point p1
        if (m & 1): output = add_points(A, B, p, p1, output)

        # Shift the bit-representation of m by 1 bit to the right and
        # double the point p1
        m >>= 1
        p1 = add_points(A, B, p, p1, p1)

    return output
```

5. **is_on_ec(A, B, p, p1)**: it checks if a given point $p1$ belongs to the elliptic curve group associated with the elliptic curve equation $E : y^2 = x^3 + Ax + B \pmod p$

```
def is_on_ec(A, B, p, p1):
    if (p1 == "0"): return True

    x_p1, y_p1 = p1
    return ((y_p1 ** 2)%p == (x_p1**3 + A*x_p1 + B)%p)
```

We also saw that bitcoin's ECDSA uses the **secp256k1** elliptic curve. The following python variables specify the parameters of this curve:

- **p_dec** denotes the decimal value of the order of the underlying finite field.
- **G_dec** corresponds to the decimal coordinates modulo **p_dec** of the base point G.
- **n_dec** is the decimal value of the order of {G}, the subgroup generated by G.
- **A_dec** and **B_dec** are the decimal values modulo **p_dec** of the parameters A and B appearing in the short-Weierestrass form representation of the elliptic curve:

$$E : y^2 = x^3 + Ax + B \pmod{p}$$

  For secp256k1, **A_dec** $= 0$ and **B_dec** $= 7$

```
p_dec = long(2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1)

G_dec = (55066263022277343669578718895168534326250603453777594175500187360389116729240,
         32670510020758816978083085130507043184471273380659243275938904335757337482424)

n_dec = 115792089237316195423570985008687907852837564279074904382605163141518161494337

A_dec = 0
B_dec = 7
```

The ECDSA algorithm requires us to specify an appropriate hashing function. In the case of bitcoin, we use SHA256. In addition, the Σ algorithm makes use of a truncated version of the hash of the message. These two operations are implemented as follows:

- The method **truncate(num, L_n)** outputs the integer representation of the L_n least significant bits of the argument **num**.

```
def truncate(num,Ln):
    # Convert to binary format, remove leading 2 characters and
    # extract leftmost Ln bits
    num_bin = bin(num)[2: Ln+2]
    return int(num_bin, 2)
```

- The method **message_Hash(m)** takes a string **m**, computes its SHA256, and outputs the corresponding integer digest.

```
def message_Hash(m):
    # Transform m into byte format
    m_byte = m if isinstance(m, bytes) else bytes(m, 'utf-8')

    # Compute digest in hexadecimal format
    digest_hex = hashlib.sha256(m_byte).hexdigest()

    # Compute digest in decimal format
    digest_int = int('0x' + digest_hex, 16)

    return digest_int
```

Finally, we implement the three algorithms of the ECDSA scheme:

- **ecdsa_Key_Generate()** generates a private-public key pair (d, H) :

```
def ecdsa_Key_Generate():
    # Generate decimal version of private key d which
    # is a scalar in the field (F_n)*
    d_flag = False;
    while (d_flag == False):
        # Decimal value of random 256-bit scalar
        d = random.getrandbits(256)

        # Test if scalar is in the field (F_n)*
        d_flag = 0 < d < n_dec

    # Generate the decimal version of the public key H
    # associated with the private key d
    H = mul_scalar(A_dec, B_dec, p_dec, G_dec, d)

    return (d, H)
```

- **ecdsa_Sign(d,m)** takes private key d, message m and returns signature (r, s) :

```
def ecdsa_Sign(d, m):
    # The call to the "truncate" method is not really needed
    # since in this case, message_Hash corresponds to SHA256
    # which is already 256-bit long
    z = truncate(message_Hash(m),256);
    r, s = 0, 0;
    while ((r == 0) or (s == 0)):

        k_flag = False;
        while (k_flag == False):
            # Decimal value of random 256-bit scalar
            k = random.getrandbits(256)

            # Test if scalar is in the field (F_n)*
            k_flag = 0 < d < n_dec

        P = mul_scalar(A_dec, B_dec, p_dec, G_dec, k)
        r = P[0] % n_dec

        k_inv = inverse_of(k, n_dec);
        s = (k_inv * ((z + (d*r)) % n_dec)) % n_dec;

    return (r, s)
```

- **ecdsa_Verify(r,s,H,m)** checks validity of (r, s) on message m and publick key H:

```
def ecdsa_Verify(r,s,H,m):

    # check if the point H is actually on the curve
    if (not(is_on_ec(A_dec,B_dec,p_dec,H))):
        return False

    # Check if r and s are both elements of F_n*
    if ((r < 1) and (r > n_dec - 1) or (s < 1) or (s > n_dec - 1)):
        return False;

    z = truncate(message_Hash(m),256);
    s_inv = inverse_of(s, n_dec);
    u = (z * s_inv) % n_dec;
    v = (r * s_inv) % n_dec;
    w_1 = mul_scalar(A_dec, B_dec, p_dec, G_dec, u);
    w_2 = mul_scalar(A_dec, B_dec, p_dec, H, v)
    W = add_points(A_dec, B_dec, p_dec, w_1, w_2)

    return (r == (W[0] % n_dec))
```

We run an instance of the above algorithms as follows:

```
# Generate key pair
(d,H) = ecdsa_Key_Generate();
print "\n---------------------------   ECDSA KEY PAIR GENERATION   ---------------------------"
print "The generated private key is \n--- d = ", d;
print "The generated public key is H = (Hx, Hy) where: ";
print "--- Hx = :", H[0];
print "--- Hy = :", H[1];


# Sign message
print "\n---------------------------   ECDSA MESSAGE SIGNATURE   ----------------------------"
m = "This is a test message";
print "The signed message is m = '", m, "'";
(r,s) = ecdsa_Sign(d, m);
print "The resulting signature tuple (r,s) is given by:";
print "--- r = ", r;
print "--- s = ", s;


# Verify signature on message
print "\n---------------------------- ECDSA SIGNATURE VERIFICATION ----------------------------"
ver = ecdsa_Verify(r,s,H,m);
print "(r,s) is a ", ver, "signature on m using public key H";

r_modified = r-1;
print "r_modified is: ", r_modified;

ver = ecdsa_Verify(r_modified,s,H,m);
print "(r_modified,s) is a ", ver, "signature on m using public key H";
```

```
----------------------------------   ECDSA KEY PAIR GENERATION   ----------------------------------
The generated private key is
--- d =  73763309768538431310580271098473889669112826476412031668556512833970755797356
The generated public key is H = (Hx, Hy) where:
--- Hx = : 69156193594462789135328886659929216038700304500958935457595810271088675330427
--- Hy = : 102690096755280191023953550546049224251481987205189473743797596079104519695 1

----------------------------------   ECDSA MESSAGE SIGNATURE   ----------------------------------
The signed message is m = ' This is a test message '
The resulting signature tuple (r,s) is given by:
--- r =  25271640611405224383959318859012802138896943934637459215903900638911414159358
--- s =  36359072674130334879589777237167582672493065114074787183194150424748530149596

---------------------------------- ECDSA SIGNATURE VERIFICATION ----------------------------------
(r,s) is a  True signature on m using public key H
r_modified is:  25271640611405224383959318859012802138896943934637459215903900638911414159357
(r_modified,s) is a  False signature on m using public key H
```

ECDSA encoding In bitcoin, an ECDSA signature (r, s) is not encoded as a simple concatenation of r and s. Instead, it follows the **Distinguished Encoding Rules** or **DER** for short. Those rules are formalized in the **Abstract Syntax Notation One** standard (**ASN.1** for short) commonly used to encode arbitrary data objects into a structured binary file [14]. They allow for data compatibility between systems that may use different representations. However, the merit of using it in bitcoin remains unclear.

When (r, s) is encoded in DER format, we obtain a sequential structure of the form:

$$[A] - [B] - [C] - [D] - [E] - [F] - [G] - [H] - [I] - [J]$$

where:

- $A \equiv$ {*hex representation of the byte-length of the signature*}: This encapsulates the length of the full encoding to follow. A is allocated 1 byte.
- $B \equiv$ {*ASN.1 tag identifier for data of type "SEQUENCE"*}: This indicates that the data to follow is a constructed sequence. In ASN.1, the corresponding hexadecimal value is always 0x30. B is allocated 1 byte.
- $C \equiv$ {*hex representation of the byte-length of the* (r, s) *component*}: This encapsulates the length of the (r, s) component, inclusive of relevant ASN.1 tag identifiers as described hereafter. C is allocated 1 byte.
- $D \equiv$ {*ASN.1 tag identifier for data of type "INTEGER"*}: This indicates that the data to follow is an integer. In ASN.1, the corresponding hexadecimal value is always 0x02. D is allocated 1 byte.
- $E \equiv$ {*hex representation of the byte-length of the* r *component*}: This encapsulates the length of the r component. E is allocated 1 byte.
- $F \equiv$ {*big-endian format hex representation of the* r *value*}: This is the actual value of r represented in hexadecimal.
- $G \equiv$ {*ASN.1 tag identifier for data of type "INTEGER"*}: This indicates that the data to follow is an integer. In ASN.1, the corresponding hexadecimal value is always 0x02. G is allocated 1 byte.
- $H \equiv$ {*hex representation of the byte-length of the* s *component*}: This encapsulates the length of the s component. E is allocated 1 byte.
- $I \equiv$ {*big-endian format hex representation of the* s *value*}: This is the actual value of s represented in hexadecimal.
- $J \equiv$ {*Sighash byte*}: A one-byte specifier that we will describe when we discuss bitcoin transactions in another post.

Note that the above structure allows us to automatically deduce that:

{ $A = C+$ 0x03. The 0x03 corresponds to the 3 bytes allocated to B, C and J.

{ $C = E + H+$ 0x04. The 0x04 corresponds to the total bytes allocated to D, E, G and H.

{ Theoretically, E and H can take on any value between 1 and 33 (decimal) or equivalently 0x01 and 0x21 (hexadecimal). However, the most probable values are 32 and 33. To understand why, we first note that r and s are at most 256-bit long values by definition of bitcoin's ECDSA. However:

  - bitcoin's implementation requires that the most significant bit in the binary representation of r and s be equal to 0. As a result, if either r or s have a bit length that is a multiple of 8 i.e., of the form $8k$ for $k = 1, ..., 32$, then a most significant 0 byte is added. In this case, the size of r or s becomes equal to $(k + 1)$ bytes instead of k bytes.

  - The probability that either r or s be 33-byte long corresponds to the probability that either r or s be 256-bit long originally (i.e., $k = 8$ in the notation above). That means that we are interested in the probability of the most significant bit of a 256-bit long sequence be equal to 1. This evaluates to $\frac{1}{2}$ since bits are chosen at random.
  - The probability that either r or s be 32-byte long is equal to $\frac{255}{512}$ since it corresponds to the sum of:
    * The probability that the decimal value of the most significant byte of a 256-bit (i.e., 8 byte) long sequence be greater than or equal to 1 but less than 128. This occurs with probability $\frac{127}{256}$.
    * The probability that the most significant byte of a 256-bit (i.e., 8 byte) long sequence be equal to 0 and the first bit of the second most significant byte be equal to 1. This occurs with probability $\frac{1}{512}$.
  - So the probability that either r or s be 32 or 33 bytes long is equal to $\frac{1}{2} + \frac{255}{512} = \frac{511}{512}$. There remains a probability of $\frac{1}{512}$ for the length to be less than 32-byte long. One can calculate the probability of occurence of any of these lengths by following a similar logic to the one just outlined above.

To see an example of how this encoding is conducted in practice, consider an (r, s) signature given by its decimal representation:

$$r_{dec} =$$
$$61650733893590164207477587688588415609194348185876455223473721547793911129291$$

$$s_{dec} =$$
$$34204417344248643370177991773635004864182284445248039044681418183938145138707$$

This translates to a big-endian hexadecimal representation given by:

$$r_{hex} =$$
$$0\text{x}884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb$$

$$s_{hex} =$$
$$0\text{x}4b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813$$

- Since r_{hex} is 32-byte long with a most significant bit of 1, the bitcoin protocol mandates the addition of an extra 0 most significant byte. As such, we get:

$$E \equiv 0\text{x}21 \text{ (i.e., 33 in decimal)}$$

$$F \equiv 0\text{x}00884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb$$

- On the other hand, s_{hex} is 32-byte long with a most significant bit of 0. We get:

$$H \equiv 0\text{x}20 \text{ (i.e., 32 in decimal)}$$

$$I \equiv \text{0x}4b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813$$

- We also have $D = G \equiv$ 0x02 since they specify an integer value.
- The above allows us to compute $C = E + H+$ 0x04 $\equiv$ 0x45.
- Assume that $J =$ 0x01, i.e. the sighash byte is set to 1 (we will introduce sighash when we discuss bitcoin transactions in another post).
- By definition, $B \equiv$ 0x30.
- Finally, we calculate $A = C+$ 0x03 $\equiv$ 0x48.

As a result, the DER-encoding of (r, s) becomes:

483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae
24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e381301

Security of ECDSA: The importance of the randomness of the parameter k. A necessary condition for the ECDSA scheme to be secure is for the parameter k to be used once per each signature instance. The same logic applied earlier to DSA demonstrates that if this were not the case, one could easily retrieve the private key d of the signer. DSA's derivation can be replicated by replacing x with d and q with n.

An example that underscores the importance of k is the hacking incident that affected Sony in December 2010. At the time, a group known as "fail0verflow" successfully retrieved the ECDSA private key that signed software for PlayStation3. The reason the hackers were able to do so was because Sony misimplemented ECDSA's Σ algorithm by forcing a static k instead of choosing a random one for every signature.

Security of ECDSA: A note on existential unforgeability. There are no known proof of ECDSA's security in the RO model. This may be surprising, given ECDSA's usage in bitcoin. Here too, similarly to DSA, the belief that a security proof may be difficult to construct rests on our inability to date to successfully leverage the reduction model.

One can use the same reasoning outlined earlier for DSA to argue why a security proof for ECDSA based on the reduction model may be difficult to devise. The aforementioned logic can be applied in exactly the same way, save for a few nuances surrounding condition $\mathbf{C}_1$ that we describe next. One way of solving for an ECDSA private key is by constructing two distinct signatures $\sigma(m_1) \equiv (r_1, s_1)$ and $\sigma(m_2) \equiv (r_2, s_2)$ that lead to a linear equation in the unknown d. Conditions $\mathbf{C'}_1$ and $\mathbf{C}_2$ below are jointly sufficient for this to be possible:

$$\mathbf{C'}_1) \;\; (u_1 \otimes_p G) \; \oplus_p \; (v_1 \otimes_p H) \; = \; (u_2 \otimes_p G) \; \oplus_p \; (v_2 \otimes_p H)$$

$$\mathbf{C}_2) \;\; v_1 \neq v_2$$

Writing $H = d \otimes_p G$, condition $\mathbf{C'}_1$ becomes:

$$(u_1 + dv_1) \otimes_p G = (u_2 + dv_2) \otimes_p G.$$

Invoking $\mathbf{C}_2$ along with the fact that order$(G) = n$, we can compute:

$$d = (u_1 - u_2) \times (v_2 - v_1)^{-1} \pmod{n}.$$

In what follows, we derive necessary conditions for $\mathbf{C'}_1$ and $\mathbf{C}_2$ to hold. Since both signatures are assumed to be valid, the verification equations guarantee that:

$$Abscissa[(u_i \otimes_p G) \ \oplus_p \ (v_i \otimes_p H)] \pmod{n} \equiv x_{w_i} \pmod{n}$$

$$=$$

$$r_i \equiv Abscissa(k_i \otimes_p G) \pmod{n} \text{ , for } i = 1, 2$$

As a result:

$$\mathbf{C'}_1 \Rightarrow (r_1 = r_2) \iff (Abscissa(k_1 \otimes_p G) \pmod{n} = Abscissa(k_2 \otimes_p G) \pmod{n})$$

Consequently, k_1 and k_2 must exhibit a certain relationship for the first condition to hold. With overwhelming probabilty, two randomly chosen parameters k_1 and k_2 will not satisfy this equality.

Since $\mathbf{C'}_1 \Rightarrow (r_1 = r_2)$ and since $v_i = r_i \times s_i^{-1} \pmod{n}$, for $i = 1, 2$, we can write:

$$\mathbf{C'}_1 \cap \ \mathbf{C}_2 \ \Rightarrow \ (r_1 = r_2) \cap (s_1 \neq s_2)$$

Recalling that $s_i = k^{-1} \times (t \circ \mathcal{H}(m_i) \ + \ (d \times r_i)) \pmod{n}$, for $i = 1, 2$, we conclude that:

$$(r_1 = r_2) \cap (s_1 \neq s_2) \ \Rightarrow \ (k_1 \neq k_2) \ \cup \ (\mathcal{H}(m_1) \neq \mathcal{H}(m_2))$$

Similarly to DSA, the takeaway is that to be able to effectively use the reduction technique to solve for d in the case of ECDSA, one will **possibly** need to ensure that valid signatures $\sigma(m_1)$ and $\sigma(m_2)$ satisfy the following at a minimum:

1. $Abscissa(k_1 \otimes_p G) \pmod{n} = Abscissa(k_2 \otimes_p G) \pmod{n}$, and
2. $\mathcal{H}(m_1) \neq \mathcal{H}(m_2)$ or $k_1 \neq k_2$.

Here too, we purposely used the term "possibly". The remaining part of the argumentation is exactly the same as the one we previously outlined for DSA. We highlight again that out objective was not to argue that a security proof for ECDSA is not possible, but rather that such a proof may be difficult to achieve using the reduction technique.

Despite the absence to date of a security proof for ECDSA in the RO model, slight variants were shown to be secure:

1. The **Brickell** variant consists of replacing the function $(x, y) \longrightarrow x \pmod{n}$ that appears in ECDSA's Σ and $\mathcal{V}$ algorithms by a random oracle $\mathcal{H}_2$. In other words,

r becomes $\mathcal{H}_2(k \otimes_p G)$ and the verification algorithm checks if $r = \mathcal{H}_2(W)$. A proof of its security could be constructed in a similar way to the one provided for DSA in [5].

2. The **Pointecheval-Vaudenay-Lee-Smart** variant also known as **ECDSA - II** takes the hash of m and r combined instead of that of m alone (i.e., $\mathcal{H}(m, r)$ instead of $\mathcal{H}(m)$). This construct is similar to the DSA variant introduced earlier and we refer the reader to [10] for a proof of its security.

Aside from these variants, **Brown** [6] and Fersch et al. [8] devised two different security proofs for the unmodified version of ECDSA by introducing extra modeling constraints.

3. In the case of **Brown**, $\mathcal{H}$ is not assumed to behave as a random oracle. However, the underlying group is modeled as a *generic group*. The *generic group* assumption is a strong one since it was shown in [13] that it implies that ECDSA would be strongly unforgeable (and hence non-malleable), a conclusion that is known not to be valid for ECDSA as we discuss in a subsequent section. We will not go over the details of generic groups or Brown's model, but refer the interested reader to [6].

4. In the case of Fersch et al., $\mathcal{H}$ is assumed to be a random oracle. In addition, the authors impose a constraint (similar to the one imposed in their proof for the DSA case), known as the **bijective random oracle**. This constraint is applied to the **conversion function** which in the case of ECDSA is defined as:

$$f : (\mathbb{F}_p \times \mathbb{F}_p) \longrightarrow \mathbb{Z}_n^*$$

$$e \longrightarrow Abscissa(e) \pmod{n}$$

The conversion function is none else than the one that ECDSA's Σ algorithm uses to calculate r with elliptic curve point P as input. The constraint that Fersch et al. impose consists of representing f as a composition of three functions $\psi \circ \Pi \circ \phi$, such that Π is a bijection and such that both Π and Π^{-1} are modeled as random oracles. We will not go over the details of their proof, but the interested reader can refer to [8].

Security of ECDSA: Signature's malleability. Once a signature $\sigma(m)$ has been issued on a given message m, it is reasonable to require that no adversary be able to devise another valid signature $\sigma'(m) \neq \sigma(m)$ on the same message. A signature is said to be **malleable** if it is not subjected to the aforementioned requirement. As a result, signature malleability could potentially lead to an instance of forgery, albeit in a restrictive sense since the message is taken to be the same. On the other hand, signatures that are simulteneously not malleable and existentially unforgeable (i.e., resilient against EFACM) are referred to as **strongly unforgeable** [4].

Signature malleability leads to **transaction malleability**, a notion that we will discuss in a separate post dedicated to bitcoin transactions. For the purpose of our current discussion, it suffices to highlight that a bitcoin transaction is a data structure that encompasses four main categories of information:

1. Source(s) of funds to be transfered, also known as unspent transaction output(s) or UTXO(s).
2. Destination(s) of the funds (i.e., the intended recipient(s) addresse(s)).
3. Exact amount to be sent to each destination address.
4. Information containing the DER-encoded ECDSA signature(s) on the relevant UTXO(s).

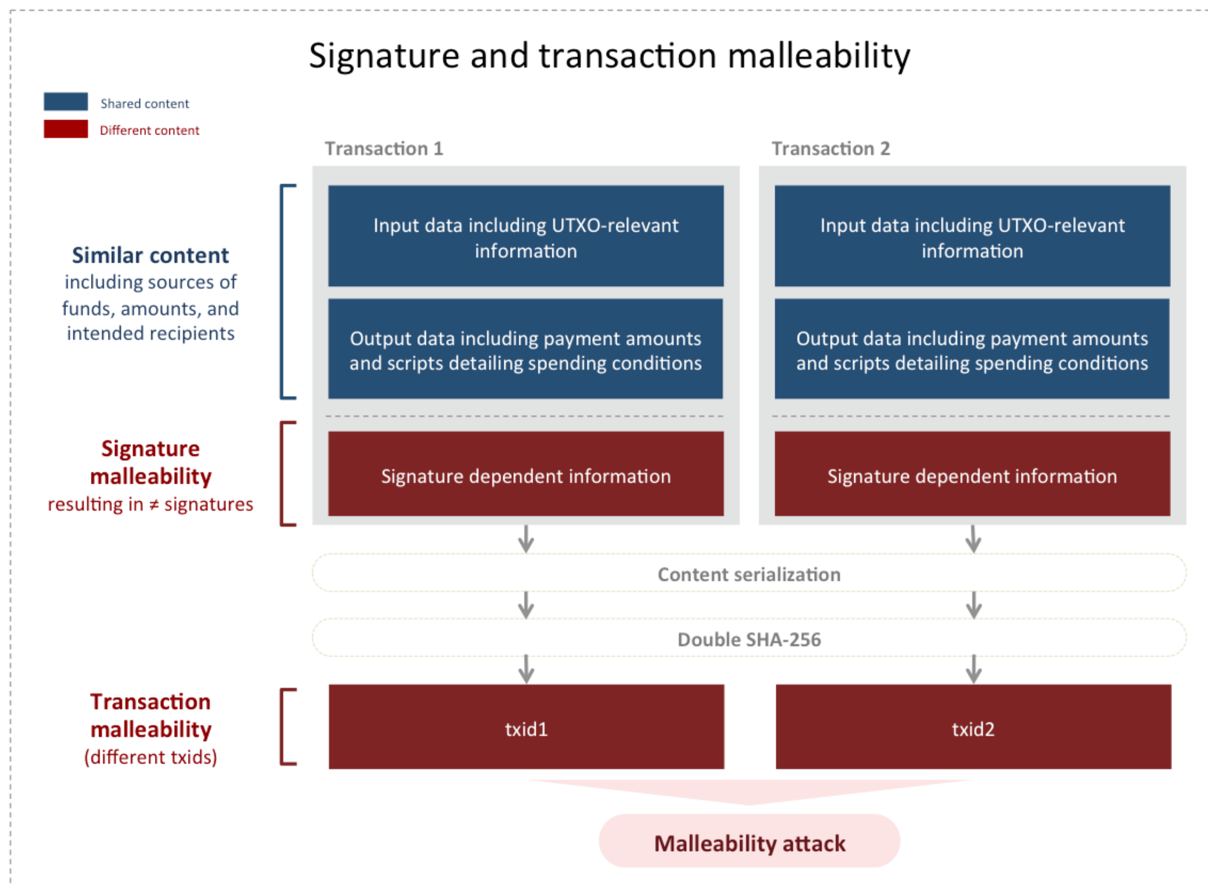
Transactions are represented in a serialized byte format that we discuss in more detail in the bitcoin transactions post. The raw serialization is subjected to a double SHA-256 operation that outputs a hexadecimal digest known as the transaction id or **txid** for short. Any alteration to the body of the transaction, no matter how small, results in a different txid. This is a direct consequence of the expected behavior of a hashing function.

The critical observation is that although a signature is part of the body of the transaction, it is logically infeasible to sign a data structure inclusive of the resulting signature itself. Instead, the signing process is applied to the content of the transaction exclusive of the signature. More specifically, the message that gets signed includes information about the funding UTXOs, the destination addresses and their respective intented amounts.

By definition, a malleable signature scheme could lead to the creation of two valid but different signatures applied to the same transaction. Such an event would cause the bitcoin network to end up with at least two different txids referencing the same content. Such a situation could motivate a specific type of attack known as a **malleability attack**. The gist of it is as follows:

1. Suppose Alice issues a BTC payment to Bob. Let $txid_1$ be its transaction id.
2. Suppose that Bob alters the signature of Alice's transaction (assuming it is a malleable scheme) right before $txid_1$ gets any confirmation on the blockchain. This alteration results in a new transaction id, namely $txid_2$, on the same content (i.e., the intended recipient is still Bob, the funding UTXOs are still the same, and the amount remains as is).
3. If $txid_2$ gets confirmed on the blockchain before $txid_1$, the latter will become orphaned. If Alice does not have the required level of sophistication to track UTXOs on the blockchain in order to verify that her original UTXOs have been spent, it will rely instead on the confirmation status of $txid_1$. Given that it was orphaned, it will conclude that the funds never reached Bob's address.
4. Bob could then defraud her by asking her to issue a new payment knowing that he would have already received the intended funds by virtue of $txid_2$ being confirmed. He would then receive twice the intended amount.

The above malleability attack can be interpreted as an instance of double-spending, although the malicious party in this case is the receiver and not the sender.



It turns out that ECDSA is malleable. In what follows, we describe three possible avenues to change it without modifying relevant content in the transaction. We highlight that the first two avenues could be exploited by any party including e.g., the recipient of a given transaction. As a result, they are conducive to malleability attacks. On the other hand, the third avenue is specific to the holder of the private key. If the sender is the only holder of the key, one can reasonably assume that no malleability attacks would ensue. We point out that bitcoin has already implemented measures to prevent the first two avenues from being nefariously exploited:

1. Malleability caused by **Non-DER encoded ECDSA signatures**: We described earlier how to encode an ECDSA signature into DER format. Given an (r, s) pair, one can see that by diligently applying the DER encoding procedure, the resulting output will be unique. In particular, a strict implementation of DER would not allow prepending any number of 0 bytes to the octet representation of integers. The only exception occurs if the most significant bit of this octet representation is equal to 1, in which case we prepend a single 0 byte. For example:

   - An r value of 0x4b9 cannot be encoded as e.g., 0x004b9.
   - An s value of 0x884d must be encoded as 0x00884d (since the most significant bit is equal to 1), but cannot be encoded as e.g., 0x0000884d.

   However, bitcoin's original implementation did not strictly enforce this rule. As a result, one could derive an infinite number of encodings for a given (r, s) pair.

This source of signature malleability has been addressed in Bitcoin Improvement Protocol 66 (BIP 66) [15].

2. Malleability caused by **ECDSA's inherent signature construct**: Given an ECDSA signature $\sigma(m) \equiv (r, s)$, one can automatically devise another valid signature on m by replacing s with $(n - s)$, where n is the order of G. In other terms, $\sigma'(m) \equiv (r, n - s)$ is a valid signature on m. To see why, recall that $\sigma(m)$ satisfies the verification algorithm $\mathcal{V}$ which consists of the following steps:

   - Compute $u = [t \circ \mathcal{H}(m)] \times s^{-1} \pmod n$
   - Compute $v = r \times s^{-1} \pmod n$
   - Compute $W \equiv (x_w, y_w) = (u \ \otimes_p \ G) \ \oplus_p \ (v \ \otimes_p \ H)$
   - Validate that $r = x_w \pmod n$

   On the other hand, running the verification algorithm $\mathcal{V}$ on $\sigma'(m)$ will:

   - Compute $u' = [t \circ \mathcal{H}(m)] \times (n - s)^{-1} \pmod n$
   - Compute $v' = r \times (n - s)^{-1} \pmod n$
   - Compute $W' \equiv (x_{w'}, y_{w'}) = (u' \ \otimes_p \ G) \ \oplus_p \ (v' \ \otimes_p \ H)$
   - Check if $r = x_{w'} \pmod n$

   Let $a \equiv (n - s)^{-1} \pmod n$ We get the following implications:

$$a \equiv (n - s)^{-1} \pmod n \iff a \times (n - s) = 1 \pmod n$$

$$\iff a \times n - a \times s = 1 \pmod n \iff -a \times s = 1 \pmod n$$

$$\iff -(s)^{-1} = (n - s)^{-1} \pmod n$$

   As a result, we rewrite $\sigma'(m)$'s verification steps as follows:

   - Compute $u' = [t \circ \mathcal{H}(m)] \times (-s^{-1}) \pmod n = -u \pmod n$
   - Computing $v' = r \times (n - s)^{-1} \pmod n = r \times (-s^{-1}) \pmod n = -v \pmod n$
   - Computing $W' \equiv (x_{w'}, y_{w'}) = (u' \ \otimes_p \ G) \ \oplus_p \ (v' \ \otimes_p \ H) = -W$. Recall that as was introduced in the *Elliptic Curve Group* post, the inverse of the elliptic curve point $W \equiv (x_w, y_w)$ is $-W \equiv (x_w, -y_w \pmod n))$. And so $x_w = x_{w'}$.
   - Since $\sigma(m)$ is a valid signature, we have $r = x_w \pmod n$. In light of the previous equality, we deduce that $r = x_{w'} \pmod n$ and as a result, that $\sigma'(m)$ is also a valid signature on m.

   This type of signature malleability was supposed to be addressed in BIP62, but had to wait until Pull Request #6769 [1] to be resolved. The mitigation mechanism consisted of requiring that only the signature with the lowest s value be valid.

3. Malleability caused by **ECDSA's reliance on the random parameter** k: The third source of signature malleability is a direct result of the presence of the

parameter k in the signing algorithm Σ. A signer could decide to sign the message as many times as she likes, and the randomly generated parameter k will ensure that these signatures are different. However, as mentioned earlier, this source of malleability does not lend itself to attacks of the type previously described.

ECDSA multisignatures. So far, our discussion of ECDSA signatures was limited to single signers. It turns out that more elaborate signatures could be constructed. In particular, we could have $n > 1$ private keys jointltly sign a transaction in what is commonly known as a multisignature. An important observation is that the implementation of multisignatures in bitcoin consists of creating a separate signature for each private key and then grouping these signatures together. This construct results in at least three disadvantages:

1. **Size inefficiency**: The multisignatures size grows linearly with the number of signers. A direct impact is that multisignatures will occupy a bigger portion of a block on the blockchain. And since block issuance is kept constant at ∼1 block per 10 minutes, this results in a lower transaction throughput and slower processing.
2. **Higher cost**: Transaction fees in bitcoin are calculated on a kilobyte basis. This implies that larger transaction sizes will cost more. As a result, multisignatures are more costly than monosignatures.
3. **Less privacy**: In order to verify the validity of an ECDSA multisignature, the network must have access to the set of the public keys of the signers. Making the public key set known will signal that the transaction is of a multisignature type and could draw unecessary attention from potential attackers. Ideally, one would want to keep private the multisignature nature of a given transaction.

In conclusion, we note that despite the ECC-inherited security features of ECDSA, the signature scheme is not fully immune to drawbacks including:

1. An absence to-date of a proof of ECDSA's security in the RO model. For some, this may be a non-issue, but others would prefer to use a scheme that is at least provably secure in this idealized setting.
2. An inherent malleability buit in ECDSA signatures.
3. A size-inefficient, more costly and less private implementation of multisignatures.

In light of these shortcomings, a new BIP advocating the adoption of a different signature scheme has been put forth. It turns out that similar to the Schnorr scheme, a variant of it known as Elliptic Curve Schnorr [16] is provably secure and non-malleable in the RO model. Moreover, this variant benefits from the linearity property that allows multiple private key holders to jointly sign a transaction such that the resulting signature is not a naive concatenation of individual signatures, but rather a non-trivial aggregation that reduces to a monosignature. We will discuss multisignatures and explain the advantages of the Elliptic Curve Schnorr variant in a separate post.

References

[1] Pull request 6769 - script verify low s. https://github.com/bitcoin/bitcoin/pull/6769, 2015.

[2] Anonymous. Rebutal to schnorr's patent claims re dsa. https://bit.ly/2SrEAeW, August 1998.

[3] Elaine Barker. Recommendation for key management part 1. *NIST Special Publication 800-57 Part 1 Revision 4*, January 2016.

[4] D. Boneh, E. Shen, and B. Waters. Strongly unforgeable signatures based on computational diffie-hellman. *PKC LNCS*, 3958:229–240, 2006.

[5] Ernest Brickell, David Pointcheval, Serge Vaudenay, and Moti Yung. Design validations for discrete logarithm based signature schemes. *Public Key Cryptography. PKC 2000. Lecature Notes in Computer Science*, 1751, 2000.

[6] Daniel R.L. Brown. The exact security of ecdsa. *Technical Report CORR 2000-34 Certicom Research*, 2000.

[7] Andrea Corbellini. Elliptic curve cryptography, a gentle introduction. http://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/.

[8] Manuel Fersch, Eike Kiltz, and Bertram Poettering. On the provable security of (ec)dsa signatures. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1651–1662, October 2016.

[9] David W. Kravitz. Digital signature algorithm patent. https://patents.google.com/patent/US5231668, 1991.

[10] John Malone-Lee and Nigel P. Smart. Modifications of ecdsa. *Selected Areas in Cryptography—SAC*, 2595:1–12, 2003.

[11] David Pointcheval and Serge Vaudenay. On provable security for digital signature algorithms. 11 1996.

[12] Claus P. Schnorr. Method for identifying subscribers and for generating and verifying electronic signatures in a data exchange system. https://patents.google.com/patent/US4995082, 1989.

[13] J. Stern, D. Pointcheval, J. Malone-Lee, and N.P. Smart. Flaws in applying proof methodologies to signature schemes. *CRYPTO*, pages 93–110, 2002.

[14] International Telecommunication Union. Itu-t x.690. https://www.itu.int/rec/T-REC-X.690/, July 2002.

[15] Pieter Wuille. Bip66 - strict der signatures. https://github.com/bitcoin/bips/blob/master/bip-0066.mediawiki, 2015.

[16] Pieter Wuille. proposed bip for 64-byte elliptic curve schnorr signatures. https://github.com/sipa/bips/blob/bip-schnorr/bip-schnorr.mediawiki, July 2018.