Security Aspects of Bitcoin

Erik Vesterlund

June 9, 2016

Abstract

Digital signatures (ECDSA) and hash functions (SHA-256) are used to secure ownership and transactions of the cryptocurrency Bitcoin. We describe the strengths and vulnerabilities of these technologies, and motivate where and why they appear in the Bitcoin protocol.

Contents

1 The Public-Key Paradigm

Bitcoin is a digital currency which relies on cryptography. Creating transactions, verifying them and making them part of the block chain - Bitcoin's public distributed ledger containing all transactions ever made - is done entirely by those using the currency, without the need for a trusted third party. To make this possible, Bitcoin makes use of one-way functions and trap-door functions; we begin with the latter type.

In private-key symmetric cryptography, parties agree on a common, secret key by which to encrypt and decrypt messages between each other. While two parties could physically meet to agree on the common key, this is rarely feasible whence secure communications would have to be established. In this once exclusive paradigm, the parameters of secure communication would have to be kept absolutely secret, and if an adversary would have acquired said parameters the security of communication would be compromised.

In 1976, Whitfield Diffie and Martin Hellman (DH) introduced a way to exchange keys over an insecure channel [13, p. 66]. The DH key exchange algorithm goes as follows: two parties, Alice and Bob, agree on a large prime number p and a parameter g and then choose private keys a and b, respectively. Alice then sends Bob the number $g^a \mod p$ and Bob sends $g^b \mod p$. Then their secret key is $g^{ab} \mod p$.

While all the parameters (g, g^a, g^b) and the algorithm used for key creation are public, or are assumed to be public ("the enemy knows the system"), an adversary cannot feasibly determine g^{ab} from this information. This is called the (computational) Diffie-Hellman problem (DHP).

1.1 Discrete Logarithms

Informally, we say that a problem is infeasible or intractable if there exists no known algorithm that can solve the problem within a reasonable time frame, where "reasonable" can mean anything from milliseconds to a few days. The solution to a problem may be sought by the straightforward way of trying various inputs until a solution is found, which is called an exhaustive or brute-force search, or some property of the problem-setting may be exploited which solves the problem faster.

All widely used cryptosystems are susceptible to brute-force attacks since each system can yield only finitely many keys. Thus, to prevent brute-force attacks the number of possible keys must be very large. We say that a cryptosystem has an n-bit security level if an attack requires at least 2^n steps (e.g. 2^n function evaluations), where n should be at least 128. [10, p. 36]

The DHP is related to the discrete logarithm problem (DLP): let $G = \langle g \rangle$ be a cyclic group of order n. Given $h \in G$, find x such that $g^x = h$. Clearly the DHP can be solved if the DLP can be solved (given g, g^a, g^b, calculate a from g^a and then calculate $(g^b)^a$). The DLP can of course be solved by a brute-force search in at most p steps, but a faster general-purpose algorithm can solve it in $\sqrt{p} \log p$ steps. This is, however, still too slow, and the DLP remains unbroken. [13, p. 78]

In the subsequent sections we describe encryption and signature schemes based on the DLP, but other cryptosystems may use different problems, e.g. the RSA cryptosystem relies on the infeasibility of factoring large integers.

1.2 Public-Key Encryption

While encryption doesn't enter into the Bitcoin protocol, we briefly outline the DLP-based Elgamal public key encryption algorithm for the sake of completeness and context.

The purpose of encryption is to provide privacy, to make messages illegible to outside partners. In public key encryption, Alice has a private key and a public key. The public key is broadcast for everyone to see, and allows anyone who wants to communicate privately with Alice to encrypt messages using her public key. Alice then uses her private key to decrypt such messages.

Generalized Elgamal encryption entails calculations in a finite cyclic group G, for which the DLP is assumed to be intractable. Let $|G| = n$ and g generates G, i.e. $G = \langle g \rangle$. Alice chooses a private key $v \in \mathbb{Z}_n^*$ and generates her public key u as $u = g^v$.

Bob, who wants to write Alice a message, obtains u. The message m must be represented as an element of G. Bob then chooses a random number $k \in \mathbb{Z}_n^*$ and computes $(c_1, c_2) = (g^k, mu^k)$ which he sends to Alice.

Alice then uses her private key to compute $c_1^{-v} c_2$. This will yield the original message since $c_1^{-v} c_2 = (g^k)^{-v} m (g^v)^k = m$ [16, p. 297].

Example. *Alice and Bob agree on the group $G = \mathbb{Z}_{541}^*$ with generator $g = 10$ (calculations are modulo 541). Alice chooses $v = 5$ as her private key and sends Bob her public key $u = 10^5 = 456$. Bob converts his message to the number $m = 357$ and chooses the random number $k = 151$. He then computes $c_1 = 2^{151} = 267$, $\delta = 35 \cdot 13^{151} = 434$ and sends to Alice.*

Alice computes $c_1^{-v} c_2 \mod p$, which comes out to the original message.

The random key k must be different for each message. Otherwise, suppose we have messages with corresponding cipher pairs (m_1, c_1, c_2) and (m_2, d_1, d_2). Then $c_2/d_2 = m_1/m_2$, and m_2 could be easily computed if m_1 were known [16, p. 296].

1.3 Signature Schemes

A digital signature of a message is a number dependent on some secret, known only to the signer, and possibly on the content of the message [16, p. 427]. Like encryption, a digital signature scheme makes use of a private key and public key, but in the case of digital signatures the private key is used to sign messages while the public key is used to verify the signature. There are many applications of digital signatures but the immediate one implied by the name is identification. Clearly, it should be easy to verify a signature but infeasible to forge a signature.

In general, a digital signature scheme consists of two parts, a signing algorithm S and a verification algorithm V. The author of a message m, Alice, generates a private signing key v and a corresponding public verification key u.

Note that, by itself, a digital signature scheme is only concerned with identification and not privacy. Creating a signature does not in any way encrypt the message; for that purpose an encryption scheme as described above must be used.

Hash functions are necessary in digital signature schemes and are described in detail in another section. For now we need only think of them as "fingerprints", non-invertible functions mapping arbitrary inputs to the non-negative integers.

1.4 The Elgamal Signature Scheme

To get more concrete, we now describe the generalized Elgamal signature scheme. Alice agree on a finite group $G = \langle g \rangle$ of order n, and Alice chooses a private signing key $v \in \mathbb{Z}_n^*$ and publishes her verification key $u = g^v$. To sign a message m, she chooses a random secret key $k \in \mathbb{Z}_n^*$ and computes $r = g^k$, $h(m)$ and $h(r)$, where h is a hash function (we needn't apply h to r if $G \subset \mathbb{Z}_n$). Finally she computes $s = k^{-1}(h(m) - vh(r)) \mod n$. Alice's signature for m is (r, s).

To verify the signature, Bob obtains the verification key and computes $t_1 = u^{h(r)} r^s$ and $t_2 = g^{h(m)}$. The signature is accepted if and only if $t_1 = t_2$. [16, p. 458]

Example. *Let $G = \mathbb{Z}_{997}^*$ generated by 2 (calculations are modulo 997). Alice chooses the private key $v = 674$ and publishes her public key $u = 2^{674} = 27$. To sign her hashed message $h(m) = 35$, Alice generates the secret key $k = 245$. She computes $r = 2^{245} = 197$ and $s = 245^{-1}(35 - 674r) = 629$, and sends Bob the information $(m, 197, 629)$.*

Bob receives the message and computes $t_1 = u^r r^s = 749$, $t_2 = 2^{35} = 749$. *Since the two are equal he accepts the message as being authored by Alice.*

As with Elgamal encryption, the random key k must be different for each message. Suppose k is used for two messages m_i with signature (r, s_i), $i = 1, 2$. Then $s_1 - s_2 = k^{-1}(h(m_1) - h(m_2))$ and we can derive k if $s_1 - s_2 \neq 0$. Once k is known we obtain the private key as $v = h(r)^{-1}(h(m_i) - s_i k)$. [16, p. 455]

1.5 Elliptic Curves

An elliptic curve E over the field k is the set of solutions to an equation of the form

$$y^2 = x^3 + ax + b \quad (a, b \in k)$$

We require that E is non-singular, i.e. it doesn't have any cusps or self-intersections. We will concern ourselves only with elliptic curves over fields of characteristic different from $2, 3$; the non-singularity requirement is then equivalent to the discriminant $\Delta = -16(4a^3 + 27b^2)$ being nonzero. [17, p. 45]

We can turn an elliptic curve into an abelian group by defining an "addition" operation on it and introducing an additional "point at infinity". Basically, if P_1, P_2 are points on the elliptic curve E we obtain their sum $P_1 + P_2$ by first drawing a line L through them and finding the third point of intersection $R = (x, y)$ of E and L; we then reflect R across the x-axis and define $P_1 + P_2 = -R = (x, -y)$. This geometric formula doesn't cover all possible cases that arise, however, so we now define addition in a more rigorous manner.

Let $P_i = (x_i, y_i)$, $i = 1, 2$, be points on the curve $E : y^2 = x^3 + ax + b$. Draw a line L through P_1, P_2, which will intersect E in a third point $P_3 = (x_3, y_3)$. Let $L : y = \lambda x + m$ with $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$ and $m = y_1 - \lambda x_1$. Substituting L into E we get

$$(\lambda x + m)^2 = x^3 + ax + b \Leftrightarrow x^3 - \lambda^2 x^2 + (a - 2\lambda m)x + (b - m^2) = 0 \quad (1)$$

Since the x^2-coefficient of $(x - x_1)(x - x_2)(x - x_3)$ is $-(x_1 + x_2 + x_3)$ we obtain $x_3 = \lambda^2 - (x_1 + x_2)$, and from the equation of the line, $y_3 = \lambda x_3 + m = \lambda(x_3 - x_1) + y_1$. We then define $-P_3 = (x_3, -y_3)$ and $P_1 + P_2 = -P_3$.

If $P_1 = P_2$, we instead define $\lambda = \frac{3x_1 + a}{2y_1}$, obtained by taking the implicit derivative of E and evaluating at P_1. To define $(x, y) + (x, -y)$ we turn to the aforementioned point at infinity, which we denote by a zero. To fully understand the point at infinity we would have to delve deeper into geometry

than is necessary for an introduction to elliptic curve cryptography; for our purposes we need only know that $(x, y) + (x, -y) = 0$ for any choice of $(x, y) \in E$, and that $0 \in E$.

Finally, in adding P_1, P_2, if one of them, say, P_1, is the point of infinity, we define $P_1 + P_2 = P_2$. We have thus made the following definition: [13, p. 285]

Definition. *Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points on E.*

a) If $P_1 = 0$, define $P_1 + P_2 = P_2$.

b) If $P_2 = 0$, define $P_1 + P_2 = P_1$.

c) If $x_1 = x_2$ and $y_1 = -y_2$, define $P_1 + P_2 = 0$.

d) Otherwise define $P_1 + P_2 = (\lambda^2 - x_1 - x_2, \lambda(x_1 - x_3) - y_1)$ where

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P_1 \neq P_2 \\ \frac{3x_1^2 + a}{2y_1} & \text{if } P_1 = P_2 \end{cases} \tag{2}$$

Save for the associativity of the above operation, it is quite obvious that it turns E into an abelian group, with the point at infinity as identity, and we denote such a group by $E(k)$. [17, p. 51] In other words, $E(k)$ denotes the points in k^2 which satisfy E's equation, along with the point at infinity.

Example. *Let $E : y^2 = x^3 - x + 1$ over $k = \mathbb{Q}$. Then $\Delta = -16(4(-1)^3 + 27 \cdot 1^2) \neq 0$, so E is an elliptic curve. The points $P_1 = (-1, 1)$, $P_2 = (0, 1)$ are easily verified as points on E. We get $\lambda = 0$ and $x_3 = -(-1 + 0) = 1$ and $y_3 = -1$, thus $P_1 + P_2 = (1, -1)$.*

To find $2P_1 = P_1 + P_1$, we first obtain the implicit derivative of E, $2ydy = (3x^2 - 1)dx$, and evaluate at P_1 to obtain

$$\lambda = \frac{3(-1)^2 - 1}{2 \cdot 1} = 1$$

Then $2P_1 = (1 - 2(-1), (-1 - 3) - 1) = (3, -5)$.

In cryptography elliptic curves are defined over a finite field. The above calculations are still valid if we work in a finite field of characteristic greater than three.

If the ground field k is finite then clearly so are the points on an elliptic curve over k. If $|k| = q$ then for any choice of $x \in k$ there are at most two y-values such that (x, y) satisfies the elliptic curve equation. Thus the number of points on an elliptic curve is, along with the point at infinity, at most $2q + 1$. Fortunately there is a much better approximation: [13, p. 289]

Theorem. *(Hasse) Let E be an elliptic curve over the field k of q elements. Then $|E(k)| = p + 1 - t_p$, where $|t_p| \leq 2\sqrt{q}$.*

Since $E(k)$ is finite for finite k, each point $P \in E(k)$ has finite order, i.e. a number n such that $nP = 0$. The elliptic-curve version of the DLP is to find the integer m such that for two given points $P, Q \in E(k)$ one has $mP = Q$.

1.6 The Elliptic Curve Digital Signature Algorithm

We can now describe the Elgamal digital signature algorithm in the setting of elliptic curves. Alice and Bob must first agree on a ground field k, a curve E, and a base point $g \in E(k)$ of prime order n, as well as a hash function h. They then follow these steps: [14, p. 135]

1) Alice chooses a secret integer v and computes the point $u = (u_x, u_y) = vg \in E(k)$. Alice's private signing key is v, and her public verification key is u.

2) Alice chooses a random key $1 < e < n-1$ and calculates the point $a = eg$, as well as $r = a_x \mod n$ and $s = e^{-1}(h(m) + rv) \mod n$; if either r or s equals zero, repeat the step with a different e.

3) Bob verifies Alice's signature (r, s) for the message m by first calculating $v_1 = h(m)s^{-1} \mod n$ and $v_2 = rs^{-1} \mod n$ and then checking that the x-coordinate of $v_1 g + v_2 u = r \mod n$.

Example. *Let $E : y^2 = x^3 - x + 1$ over the field k with 107 elements. The point $g = (106, 1)$ has prime order $n = 53$, the following calculations are modulo n.*

Alice chooses 42 as her private key, which generates her public key $(85, 17)$. For the hashed message $h(m) = 5$ she chooses the ephemeral key $e = 49$, which gives $a_x = 35$. Then her signature on $h(m)$ is $r = 35$ and $s = 49^{-1}(5 + 35 \cdot 42) = 13(5 + 39) = 42$.

Bob receives the message and $(h(m), r, s) = (5, 35, 42)$. He verifies that hashing the message produces $h(m)$, and that r is not zero. He then calculates $v_1 = 5 \cdot 42^{-1} = 5 \cdot 24 = 14$, $v_2 = 35 \cdot 42^{-1} = 35 \cdot 24 = 45$ and the point $14g + 45u = (35, 83)$, and verifies that its x-coordinate equals 35. Since all tests pass, Bob accepts the signature.

2 Hash Functions

2.1 Formatting

A hash function is a function mapping strings of arbitrary length to fixed-length bit strings, called hash values or simply hashes, and that is easy to compute. We will often refer to hash function inputs as messages, by which we mean any information representable in binary. The number of possible inputs for the hash functions we will be considering is finite but is so large that we may as well think of it as being infinite. The number of possible outputs is also quite large but nowhere near as large as the number of inputs, thus hash functions are many-to-one (surjective).

Since hash functions work with binary strings, text messages must first be converted to that format. The most common character encoding used is UTF-8 wherein the most common characters are represented by 8-bit strings. The characters most commonly seen in standard (English) texts are represented by the numbers 32 to 122. For example, the number zero is represented by $48 = \texttt{00110000}$, the letter 'A' by $65 = \texttt{01000001}$ and the letter 'a' by $97 = \texttt{01100001}$.

The output of most hash functions ranges between 128 and 1024 bits. For a more concise representation of the output, the hexadecimal (base 16) system can be used, which uses the numbers 0-9 as well as the letters a,b,c,d,e,f to represent the numbers 10-15. Each hexadecimal number represents four bits. We'll write inputs and outputs in hexadecimal whenever possible.

For the remainder of this section we reserve the labels h and n for hash functions and the number of output bits, respectively, and let $||$ denote concatenation. A string of k zeros is written as 0^k (and similar for strings of 1's).

We shall later see a number of operations on binary strings: logical 'and', denoted $\wedge$, returns 1 if both inputs are 1 and zero otherwise; exclusive 'or', denoted $\texttt{xor}$, returns 1 if exactly one input is 1 and zero otherwise; negation, denoted $\neg$, flips 0 to 1 and 1 to 0. We'll also see two functions, circular rotation $\texttt{rot}^n$ and shift $\texttt{shr}^n$. $\texttt{rot}^n(x)$ rotates the bit at position j to position $j - n$ modulo the string length. $\texttt{shr}^n(x)$ shifts each bit n steps to the right and replaces the left-most bits with zero; this is equivalent to $x \mapsto \lfloor x/2^n \rfloor$.

2.2 Cryptographic Hash Functions

The most common uses of hash functions are with digital signatures and for data integrity. [16, p. 321] For the latter use, the hash value of a message x serves as a practically unique identifier for that message. If the message is

altered even slightly, the hash value completely changes, but since hash functions are surjective there are other messages with the same hash value as x. If x' is message different from x such that $h(x') = h(x)$, then if (r, s) is a digital signature for $h(x)$ it is also a signature for $h(x')$. For cryptographic purposes, then, a hash function should be chosen that has the property that it's difficult to find an x' as described. This property is called second-preimage resistance.

More generally we'd want it to be difficult to find *any* pair x, x' with $h(x) = h(x')$. This stronger property is called collision resistance. Another important property is preimage resistance, i.e. given $h(x)$ it should be difficult to find x. A hash function that is collision resistant is almost always preimage resistant [16, p. 324], and we will assume that to be the case throughout this exposition. We then make the following informal definition:

Definition. *A cryptographic hash function is a function mapping arbitrary binary inputs to binary outputs of fixed length, and that is preimage and collision resistant.*

The aforementioned properties of second-preimage and collision resistance are related via the birthday paradox: in a group of people it is much more probable that two persons have the same birthday, than it is that someone's birthday is a specific date.[1] For the same reason, finding a collision is much easier than finding a second preimage. We also have the following result:

Proposition. *Collision resistance implies second-preimage resistance.*

Proof. Suppose f is a collision resistant function and fix x. If f is not second-preimage resistant then it is feasible to find x' such that $f(x') = f(x)$, whence (x, x') is a collision pair, which contradicts our assumption and the proposition follows.

Hash functions may be used along with a secret key for message authentication, in which case they are often called keyed hash functions or message authentication codes (MAC). The key k is shared with correspondents, who use it to verify that a received message x and hash value y correspond to $h_k(x) = h(k, x)$; an adversary should be unable to produce a valid MAC without knowledge of the key. [16, p. 323] As we shall see, the MAC key should be used both at the start and at the end of computing the MAC. "The" MAC, HMAC, is defined as

$$h_k(x) = h\Big((k \oplus c)||h((k \oplus d)||x)\Big)$$

[1] In a group of 23 people the probability that two persons share a birthday is about 50%, while the probability that someone has the same birthday as you is only about 6%.

where c, d are constants of equal length as block size of h. [12]

Ideally, hash functions behave as random functions, in which case finding a preimage or second preimage would take 2^n evaluations, and $2^{n/2}$ evaluations to find a collision (due to the birthday paradox); a MAC key should be large enough to make recovery infeasible. Hash functions are considered to be broken if preimages or collisions can be found through less work than advertised, even if the amount of work needed is also infeasible. [16, p. 335]

Additional desirable properties of one-way hash functions: [16, p. 331]

- Non-correlation: input bits and output bits should not be correlated, and every input bit should affect every output bit (avalanche effect).
- Near-collision resistance: it should be hard to find any two inputs whose hashes differ in only a small number of bits.
- Local one-wayness: it should be as difficult to recover any substring as to recover the entire input, and furthermore that even if part of the input is known, it should be difficult to find the remainder.

This last property is crucial to the integrity of the Bitcoin block chain since the entire input besides the nonce is known.

2.3 The Merkle-Damgård Construction

Most hash functions are designed as iterative processes which hash arbitrary-length inputs by processing successive fixed-size blocks of the input, typically 512 bit-length. [10, p. 80] Each block is then processed by a compression function f, also called round function, which in turn is composed of several other functions. This design was proposed independently by Ralph Merkle and Ivan Damgård, and is the design that was used to create the SHA-256 hash function that Bitcoin uses (and the RIPEMD-160 hash function used for generating Bitcoin addresses).

Because the round function works on blocks, the original message must be extended (padded) so that its length in bits is a multiple of the block length r. This may be accomplished by simply appending enough zeros, but this is ambiguous. Instead a single '1' is appended, followed by the needed number of zeros. (When the original input is already a multiple of r, this padding results in the creation of an extra block).

A length block may be added to the end of the padded message. For SHA-256, this is the 64-bit representation of the length of the original message. Thus the number $k \geq 0$ of zeros added (after the '1') is such that $\ell + 1 + k = 448 \mod r$, where ℓ is the bit-length of the original message.

Once the message is processed into $x_1 \cdots x_t$ consisting of t blocks, the hash value is calculated as $h(x) = f(H_{t-1}, x_t)$ where H_0 is a vector of initial values and $H_i = f(H_{i-1}, x_i)$ for $1 \leq i \leq t$. [16, p. 333]

2.4 SHA-256

In this section we described the SHA-256 hash function. [11] The input is an alphanumeric message x of bit length ℓ, and addition is modulo 2^{32}.

Let p_i be the i:th prime number. Put $H_0 = IV = (h_0, ..., h_7)$ and $K = (k_0, ..., k_{63})$, where

$$h_i = \lfloor 2^{32}(\sqrt{p_i} - \lfloor \sqrt{p_i} \rfloor) \rfloor$$
$$k_i = \lfloor 2^{32}(\sqrt[3]{p_i} - \lfloor \sqrt[3]{p_i} \rfloor) \rfloor$$

Initialize $W = (w_0, ..., w_{63}) = \mathbf{0}$.

Padding and length block:

Convert the input x to binary. Append `1` and $d \geq 0$ zeros, where d is such that $\ell + 1 + d = 448 \mod 512$. Append the 64-bit representation of ℓ. Return $x_1 \cdots x_t$ with x_{ij} the j:th 32-bit string in block x_i.

For each block x_i, $i = 1, ..., t$, do:

Message expansion:
For $j = 0, ..., 15$:
$w_j = x_{ij}$

For $j = 16, ..., 63$:
$s_0 = \texttt{rot}^7(w_{j-15}) \oplus \texttt{rot}^{18}(w_{j-15}) \oplus \texttt{shr}^{15}(w_{j-15})$
$s_1 = \texttt{rot}^{17}(w_{j-2}) \oplus \texttt{rot}^{19}(w_{j-2}) \oplus \texttt{shr}^{10}(w_{j-2})$
$w_j = w_{j-16} + s_0 + w_{j-7} + s_1$

Define $H_i = (a, b, c, d, e, f, g, h)$ and let $H_i = H_{i-1}$.

Compression loop:
For $j = 0, ..., 63$:
$S_1 = \texttt{rot}^6(e) \oplus \texttt{rot}^{11}(e) \oplus \texttt{rot}^{25}(e)$
$ch = (e \wedge f) \oplus ((\neg e) \wedge g)$
$\texttt{temp}_1 = h + S_1 + ch + k_j + w_j$
$S_0 = \texttt{rot}^2(a) \oplus \texttt{rot}^{13}(a) \oplus \texttt{rot}^{22}(a)$
$maj = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$

$$\texttt{temp}_2 = S_0 + maj$$

$$\begin{aligned}
h &= g \\
g &= f \\
f &= e \\
e &= d + \texttt{temp}_1 \\
d &= c \\
c &= b \\
b &= a \\
a &= \texttt{temp}_1 + \texttt{temp}_2
\end{aligned}$$

$$H_i = H_i + H_{i-1}$$

The hash value of x is $h(x) = a||b||c||d||e||f||g||h$. Denote the compression loop by f' so that each loop can be written $H_i = f'(H_i)$. After the first round we have $H_i = f'(H_i)$, after the second $H_i = f'(H_i) = f'(f'(H_i))$, and so on, so that in the end $H_i = (f')^{64}(H_{i-1})$. Denote the mapping $H_i \mapsto (f')^{64}(H_{i-1})$ by f, i.e. $H_i = f(H_{i-1}, x_i)$ (we include x_i here to signify that it affected the end result). We can then write $h(x)$ in the following simple way:

$$H_0 = IV;\ H_i = H_{i-1} + f(H_{i-1}, x_i);\ h(x) = H_t, \text{ or}$$

$$H_0 = IV; h(x) = H_0 + \sum_{i=1}^{t} f(H_{i-1}, x_i)$$

This representation of SHA-256 is sufficient for our purposes; while we could define the message expansion loop as $W = \Omega(x_i)$ and have f depend on W as well, in the end we're only concerned with the fact that x_i affects H_i but not precisely in what manner.

2.5 Applications

Aside from producing compressed representatives of messages, hash functions have many applications for security purposes.

2.5.1 Password Storage

Storing user passwords directly would mean that if the database where the passwords are stored is compromised, then the adversary immediately gains access to all passwords. Instead, only the hash values of the passwords should be stored, in which case the adversary would have to find (second) preimages for each hashed password of interest. [16, p. 389]

Hash functions can also be used to disguise which password in a database belongs to whom. Let p_A be the password of user A. If the database stores A and $h(p_A)$ together, an attacker would still know which hash value to invert. Suppose instead that the database stores the MAC value $h_{p_A}(A)$. Then apart from being unable either to invert the MAC or recover the password, an attacker wouldn't even know which MAC value corresponds to A's inputs.

2.5.2 Digital Signatures

As we've seen, a digital signature should be applied to the hashed message rather than the message itself, which we now justify. [16, p. 455]

In Elgamal, the signing equation without use of a hash function is $s = k^{-1}(m - vr) \mod (p-1)$, and the recipient would verify the information (m, r, s) by checking that $u^r r^s = g^m \mod p$. While not in control of the actual message, an adversary could forge a valid signature as follows: select any pair of integers e, f with $\gcd(f, p-1) = 1$ and compute $r = g^e u^f \mod p$ and $s = -rf^{-1} \mod (p-1)$. Then (r, s) is a valid signature for the message $m' = se \mod (p-1)$, since $u^r r^s = u^r (g^e u^f)^s = u^r g^{es} g^{fs} = u^r g^{m'} u^{-r} = g^{m'}$.

If a hash function is used, we could construct a hash value h' and a valid signature (r, s) for h' as above, but the recipient would not only check that $u^r r^s = g^{h'}$ but also that $h(m') = h'$, where m' is the received message. Thus the above scheme only works if we can find a preimage of h', which is infeasible if the hash function used is preimage resistant.

2.5.3 Authentication and Integrity

Hash functions find principal use for data integrity and authentication. Integrity refers to that the message m that's being sent and received remains what was intended, while authentication refers to safeguarding the origin of the message.

For the purpose of integrity, suppose a message is saved to be used at a later time. The author hashes the message so that, when accessing the message later he can hash that message and compare with the hash value he got the first time. If the hash values do not match, the message has been tampered with. [16, p. 324]

For authentication purposes hash functions can be used with or without a secret key. The message and its MAC can be sent over an unsecure channel since an interceptor would not be able to reproduce the MAC without the key. The hash value $h(m)$ alone will not do, however, so if a key isn't used $h(m)$ must be sent over a secure channel. [16, p. 364]

2.5.4 Proof of Work and Hashcash

The idea of a proof-of-work system is to allow access to a service only if the service requester can prove he's expended a certain level of work, typically computational work. Adam Back invented the Hashcash proof-of-work system in 1997 for use against email spam. In this system the sender must provide proof of work and possibly other information in, say, the subject line (header). The proof of work consists in finding a message whose hash value is small or large enough; the recipient must publicly specify which hash function to use, necessary input and the range of acceptable hash values. The range determines how much work is needed. [3]

Proof of work is used in Bitcoin in a similar fashion to update its decentralized public ledger, referred to as mining. Miners form headers x and calculate $h(h(x||\ell))$ for various integer choices of ℓ until the value is less than a specified target; the target and ℓ turn mining into a form of lottery where there will always be a "winner" (every ten minutes), but each miner's chance of success is small.

2.6 Security and Attacks

As noted, the chaining variables H_i of hash functions of the Merkle-Damgård type are calculated as $H_i = f(H_{i-1}, x_i)$, or $H_i = H_{i-1} + f(H_{i-1}, x_i)$ in the case of SHA-256, and this leads to a number of actual or potential problems.

Denote by `proc` the function taking an input to its padded and length-appended result and let h' be h where the input has already been processed, i.e. $h(x) = h'(\texttt{proc}(x))$. The security of a hash function hinges on the security of its round function f. Let h be SHA-256 and suppose $f(H_0, x_1') = f(H_0, x_1)$ where $x_1x_2 = \texttt{proc}(x)$ and $x_1'x_2 = \texttt{proc}(x')$. Then

$$h(x) = H_0 + f(H_0, x_1) + f(H_1, x_2) = H_0 + f(H_0, x_1') + f(H_1, x_2) = h(x')$$

Furthermore, suppose $\texttt{proc}(x) = x_1 \cdots x_j \cdots x_t$ and $\texttt{proc}(x') = x_1 \cdots x_j' \cdots x_t$ for some $1 \leq j \leq t-1$ (x_t is the length block) and $f(H_{j-1}, x_j) = f(H_{j-1}, x_j')$. Then $H_j = H_{j-1} + f(H_{j-1}, x_j) = H_{j-1} + f(H_{j-1}, x_j')$ so that $h(x') = h(x)$ once more. Thus if $x = x_1 \cdots x_{t-1}$ is a message whose length is a multiple of 512, a second preimage for $h(x)$ can be found if a second preimage for f as described can be found. This is a severe breach if hash functions are used without a key for authentication purposes; Alice sends Bob the message x over an unsecured channel and $h(x)$ over a secured one, but Eve intercepts x and swaps it with x' as we just described. Bob hashes x' that he's received and finds it authentic since it matches the hash value received over the secure channel.

Similar attacks could be performed to create MACs. Recall that in the environment of MACs, Alice and Bob agree on a secret key k which they hash along with the messages x they're writing each other, and transmit $(x, h_k(x))$ over an open channel. We've seen the curious HMAC construction, which we now motivate. [16, p. 355]

Many communication protocols require keys to be a specific length, and with that information Eve can actually deduce a message y and a hash value H that is a valid text-MAC pair if $\texttt{MAC}(x) = h(k||x)$.

Let ℓ_m denote the length of a message m and 64_2^m denote the 64-bit representation of ℓ_m. Suppose then that Eve intercepts a text-MAC pair $(x, \texttt{MAC}(x))$. From ℓ_k and x she knows that $\texttt{MAC}(x) = h'(k||x||1||0^d||64_2^{k||x})$ where $d = -(\ell_k + \ell_x + 1 + 64) \mod 512$. Put $p = 1||0^d||64_2^{k||x}$ and let χ be an arbitrary message (for simplicity, assume its length is < 448). Eve knows the message $y = k||x||p||\chi$ will be processed into $x_1 \cdots x_t$ as well as an extra block x_{t+1} containing χ and the padding and length of y. Eve then computes

$$H = \texttt{MAC}(x) + f(\texttt{MAC}(x), x_{t+1})$$

where $x_{t+1} = \chi||1||0^{d'}||64_2^{k||x||p||\chi}$ and $d' = -(\ell_k + \ell_y + 1 + 64) \mod 512$

Since $\texttt{MAC}(x) = H_t = H_{t-1} + f(H_{t-1}, x_t)$, when Bob receives (y, H) and calculates $\texttt{MAC}(y)$ to validate H, the MAC-value will be

$$H_{t+1} = H_t + f(H_t, x_{t+1}) = \texttt{MAC}(x) + f(\texttt{MAC}(x), x_{t+1}) = H$$

and y is accepted.

If we instead define $\texttt{MAC}(x) = h(x||k)$, then if $|x|$ is a multiple of 512, $|x'| = |x|$ and $h(x') = h(x)$ then $\texttt{MAC}(x') = \texttt{MAC}(x)$ since the MAC value will depend on the penultimate chaining variable, which will be the same due to the collision, and the final block k_p containing k and padding. Obviously this would not work if the lengths of x, x' are different multiples of 512, since the final block would be different.

Both of these issues could be resolved by using a different MAC algorithm (such as HMAC), or by modifying the hash function so that rather than setting $h(x)$ equal to the final chaining variable H_t, one would define $h(x) = g(H_t)$ for some function g. Double hashing is another solution.

3 Bitcoin

"I am convinced we shall never have good money again so long as we leave it in the hands of government. Government has always destroyed the monetary systems."

- Friedrich Hayek

The cryptocurrency Bitcoin was proposed by the pseudonym Satoshi Nakamoto in November 2008, and the first open source Bitcoin client was released shortly thereafter.

The chief contribution of Bitcoin is a way to order transactions without the need for a trusted third party. Transactions are stored in a public distributed ledger called the block chain; a block contains transactions and a pointer to the previous block, along with a partial hash-inversion, called a nonce, that is dependent on a number of information fields and validates the transactions in that block. [15, p. 11]

Users are known by their Bitcoin address or addresses, rather than by account numbers. A Bitcoin address is formed by hashing the user's ECDSA public key along with other information. [8] The ECDSA parameters used are from the secp256k1 standard; the curve used is $y^3 = x^3 + 7$ over the field of order approximately 2^{256}, with a resulting group of about the same order. [7] The address is transformed from a hash value to a 26-35 characters long string via Base58 encoding.

The two chief problems for a payment system to solve are access and double spending; only the holder of an address must be able to use the bitcoins sent to that address, and he mustn't be able to spend to the same coin twice.

3.1 Transactions

Bitcoins are created through the process of mining (to be defined later) and exist only as unspent transactions accessible to a specific user, that is, the money itself is actually just a list of transactions made to a user. A Bitcoin wallet is a program keeping track of transactions made to a user [9], and the wallet does this by checking that the user's private key can generate the recipient address of a transaction; if it can't, the transaction was made to somebody else. This scheme relies both on the intractability of the ECDLP and the irreversibility of hash functions.

Now suppose that Alice has some bitcoins and would like to make a transaction to Bob. Alice specifies how many bitcoins she wants to send,

and the wallet checks if the transactions made to her cover the transaction she wants to make. Bitcoins in a wallet are not lumped together to a large pool of bitcoins, since they only exist as transactions, so the bitcoins in the transactions that Alice wants to use, called inputs, must be spent entirely. Thus if Alice has received two transactions of three bitcoins each and wants to send Bob five bitcoins, her wallet will take both of those transactions as inputs, create a transaction to Bob of five bitcoins, and create another transaction back to Alice of one bitcoin, i.e. the change. [15, p. 10]

Apart from what transactions to draw funds from for her transaction to Bob, Alice must include her public key and digital signature for her transaction. This accomplishes two things: first, it verifies her identity, and second, others can verify that the input transactions she's using were really meant for her. Next she has to specify the outputs, consisting of Bob's address (and her own if there's any change) and the amount.

Alice has now created a complicated message whose essence is that she wants to send Bob some bitcoins. She redeemed her money by reproducing her address, acquired Bob's address and created a digital signature for her transaction to him to prove she was the one who made the transaction. The transaction is broadcast to the Bitcoin network, whose nodes verify that it was properly formed, and her inputs are marked as having been spent. [1]

However, even if the transaction was properly formed, the above security features alone don't prevent fraudulent practices. Some nodes might be malicious and refuse to update their versions of the public ledger to say that Bob now has a bit more money, but even in the case that the entire network is honest, ledgers would be updated differently simply because transactions arrive to nodes in different order.

Suppose Alice makes two transactions at virtually the same time, one back to herself and one to Bob, both transactions having the transaction `tx` as input. One node receives the transaction to Bob first, verifies it and updates his ledger to say that input `tx` has been spent, then receives the transaction from Alice to herself. This time the ledger shows that `tx` is already spent, so the transaction is denied and Bob still has the money. For another node the transactions might arrive in reverse order, so that when the transaction to Bob is to be verified, it will be referencing an already spent input, and so is denied. If Alice bought something from Bob and he's already sent the merchandise, he'll soon find that he's been cheated - according to this node.

3.2 The Block Chain and Mining

As mentioned, nodes collect transactions into blocks B_i, with each block referring to the block before it all the way back to the so-called genesis block

G. This sequence of blocks $G \leftarrow B_1 \leftarrow ... \leftarrow B_n$ is called the block chain. Now, rather than having each node update his version of the block chain on his own, nodes merely suggest blocks to the network of which only one will be accepted, thereby allowing the block chain to be the same for everyone; transactions in a block B_i are considered to have been made before the transactions in B_{i+1}, regardless of what time they were actually broadcast. [2]

What determines which suggested block will be added to the block chain is which node can first find a number ℓ, called a nonce, such that the double hash of the information in their respective block (the block header) concatenated with ℓ produces a value less than a set number (the target), i.e. who first finds ℓ such that $h^2(x||\ell) < T$; for our purposes the requirement that the hash value is less than the target can be seen regarded as requiring that the hash value begins with a certain number of zeros. If $T = 2^j$ then the probability of finding the right nonce is about 2^{j-256} [15, p. 12]; a retail computer can only try well below one billion nonces per second. [6] This process of finding the right nonce is referred to as mining. Miners are rewarded with bitcoins when they are successful (currently 25 bitcoins, halving every 210,000 blocks), and this is what creates the supply of bitcoins, which will come to and end in the next century.

Although it is highly unlikely that two blocks are confirmed at the same time, it is possible, and this creates forks from the main chain. Nodes always focus on working on the longest branch though, so once one fork becomes longer than any other the order of transactions is restored, with transactions in abandoned blocks simply re-admitted into the pool of transactions for inclusion in a later block. [15, p. 12]

This admission of transactions back into the pool again creates the possibility for double spending. Suppose Alice buys merchandise from Bob with transaction $\texttt{tx}_b$ which Bob ships to Alice once the transaction has been included into a confirmed block, and that she also created a simultaneous transaction $\texttt{tx}_a$ back to herself using the same inputs as for $\texttt{tx}_b$. Let $B_{n+1} \ni \texttt{tx}_b$ be the next confirmed block; Bob ships the merchandise. During this time Alice has simultaneously created and confirmed on her own another block $B_n \leftarrow A_1 \ni \texttt{tx}_a$. If she can confirm another block $A_1 \leftarrow A_2$ before $B_{n+1} \leftarrow B_{n+2}$ is confirmed, her fork will become the longest, causing nodes to abandon work on B_{n+2}, and the transactions in B_{n+1} (and B_{n+2}), including $\texttt{tx}_b$, are re-admitted to the pool of unconfirmed transactions. When a node will attempt to include $\texttt{tx}_b$ into the next candidate block it will be rejected since its inputs will now show as having been spent. Thus, Alice got her merchandise and Bob got zilch.

The caveat of course is that it is no small matter for Alice to do what we've

just described. The difficulty is set so that it takes the network approximately ten minutes to confirm a block, meaning that with the network's combined hash rate (approximately 2^{60} hashes per second at the time of this writing [4] it takes ten minutes to find a small enough hash value. As mentioned, a retail computer has a hash rate of less than 2^{30}, while a hash rate of about 2^{40} can be achieved with specialized hardware [5], so it is highly unlikely that Alice will outperform the network twice in a row. Despite the computational difficulty of hash inversion it is however recommended that merchants not accept purchases before six blocks have been confirmed.

References

[1] A. M. Antonopoulos, *Mastering Bitcoin*, Onlineed.,2015,http://chimera.labs.oreilly.com/books/1234000001802/index.html/

[2] The Bitcoin Wiki, https://en.bitcoin.it/w/index.php?title=Block_chain&oldid=59166

[3] The Bitcoin Wiki, https://en.bitcoin.it/w/index.php?title=Hashcash&oldid=60252

[4] The Bitcoin Wiki, https://en.bitcoin.it/w/index.php?title=Hash_per_second&oldid=54262

[5] The Bitcoin Wiki, https://en.bitcoin.it/w/index.php?title=Mining_hardware_comparison&oldid=58815

[6] The Bitcoin Wiki, https://en.bitcoin.it/w/index.php?title=Non-specialized_hardware_comparison&oldid=58167

[7] The Bitcoin Wiki, https://en.bitcoin.it/w/index.php?title=Secp256k1&oldid=55733

[8] The Bitcoin Wiki, https://en.bitcoin.it/w/index.php?title=Technical_background_of_version_1_Bitcoin_addresses&oldid=60246

[9] The Bitcoin Wiki, https://en.bitcoin.it/w/index.php?title=Wallet&oldid=60737

[10] N. Ferguson, B. Schneier, T. Kohno, *Cryptography Engineering*, Wiley Publishing, 2010.

[11] FIPS 180-4, http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[12] FIPS 198-1, http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf

[13] J. Hoffstein, Jill Pipher, J. H. Silverman, *An Introduction to Mathematical Cryptography*, Springer Verlag, 2008

[14] N. Koblitz, *Algebraic Aspects of Cryptography*, Springer Verlag, 1998.

[15] N. Koblitz, A. Menezes, *Cryptocash, Cryptocurrencies, and Cryptocontracts*, 2015.

[16] A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

[17] J. H. Silverman, *The Arithmetic of Elliptic Curves*, Springer Verlag, 2nd edition, 2009.