LIGHT · FAST · SWEET

TrustNote

TrustNote Cryptographic Algorithms (Draft)

TrustNote Institute of Technology

March 2018

Contents

1. Disclaimer

TrustNote Institute of Technology and Research & Development section hereby declare that, this package is under MIT open source software license and this software is distributed without any warranty. TrustNote Institute of Technology declares that we are **NOT** responsible for direct, indirect, incidental or consequential damages resulting from any defect, error or failure to perform. This package is **experimental** and a **work-in-progress**, use at your own risk. TrustNote R&D team can update (add/remove packages) any time without informing the users.

Contact Us

Business Enquires: foundation@trustnote.org

Technical Support: community@trustnote.org

Copyright

2. How do we select the best among nominees?

Selecting an efficient algorithm which also would be robust and secure while it can be performed very fast for many times per second on variety of devices is a very important and difficult process. Any failure or security hole will result in an irreparable damage. TrustNote Institute of Technology - R&D section criteria for selecting its project's cryptographic algorithms are:

- High security level and Robustness
- Better performance
- Most compatible one with variety of devices

So, any algorithm fits the best into these three criteria will be selected by R&D section. But it's not the end, after selecting the best available algorithm, experts in TrustNote will perform several challenging tests to verify it’s trustworthy.

2.1. Why Blake2?

Selecting an efficient, secure and fast hash function is crucial during the design process of any cryptocurrency infrastructure. A proper hash function algorithm has enough robustness so in case of encountering Collision attack, Preimage attack and etc. It should be undefeatable. A list of available hash functions presented in the table below:

Cryptographic hash functions		
HAVAL	MD5	JH
Kupyna	SHA-1	Skein
LM hash	SHA-2	Keccak
MD2	SHA-3	CubeHash
MD4	BLAKE2	ECOH
MD6	BLAKE	ECOH
MDC-2	Grøstl	FSB

Please, note that: The orders in table above are totally by randomness and this table is not sorted by any kind of factors.

A full list of available Hash functions' eBASH (ECRYPT Benchmarking of All Submitted Hashes) benchmarks available at:

- eBASH benchmarks for **Blake2b**.
- eBASH benchmarks for **Blake2s**.
- eBASH benchmarks for **results of benchmarking of all hash functions**.
- eBASH benchmarks for **full list of hash functions**

However, this is not the main reason for choosing Blake2. In the rest of this section, the primary reasons of selecting Blake2 among plenty of available hash functions will be explained.

BLAKE2 is a cryptographic hash function faster than MD5, SHA-1, SHA-2, and SHA-3, yet is at least as secure as the latest standard SHA-3. BLAKE2 has been adopted by many projects due to its high speed, security, and simplicity. BLAKE2 comes in two types (Aumasson , Neves , Wilcox-O'Hearn, & Winnerlein , 2017)

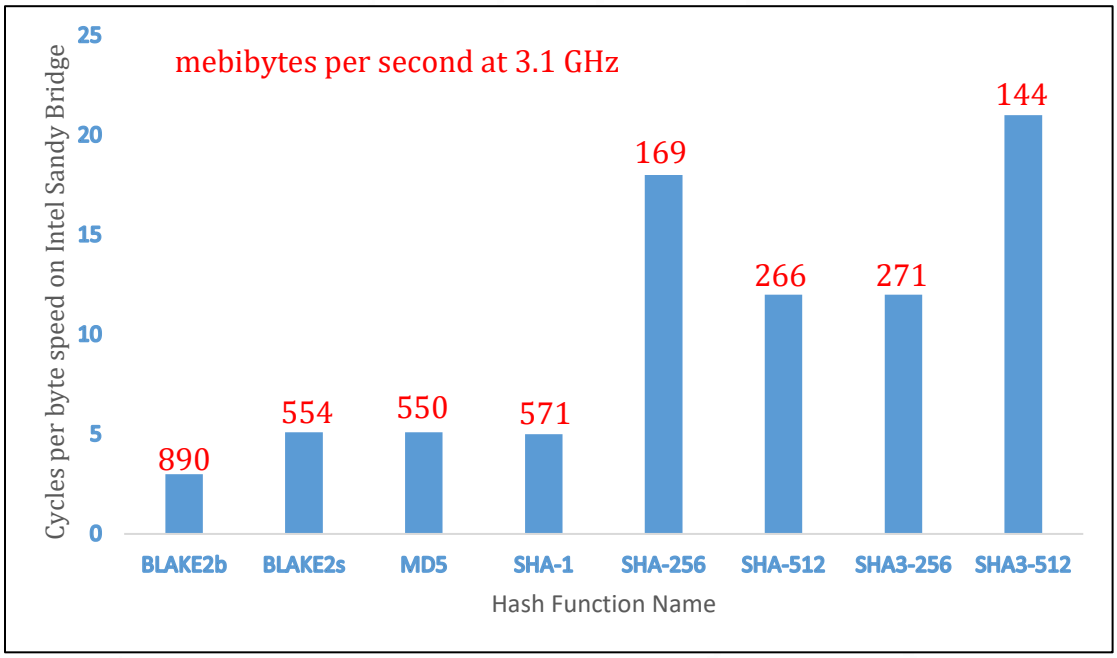
- **BLAKE2b** (or just BLAKE2) is optimized for 64-bit platforms including NEON-enabled ARMs and produces digests of any size between 1 and 64 bytes.
- **BLAKE2s** is optimized for 8- to 32-bit platforms and produces digests of any size between 1 and 32 bytes.

BLAKE2 includes the 4-way parallel BLAKE2bp and 8-way parallel BLAKE2sp designed for increased performance on multicore or SIMD (Single Instruction, Multiple Data) CPUs. BLAKE2 offers these algorithms tuned to your specific requirements, such as keyed hashing (that is, MAC or PRF), hashing with a salt, updatable or incremental tree-hashing, or any combination thereof (Aumasson , Neves , Wilcox-O'Hearn, & Winnerlein , 2017).

The SHA-3 Competition succeeded in selecting a hash function that complements SHA-2 and is much faster than SHA-2 in hardware (Chang, et al., 2012). There is nevertheless a demand for fast software hashing for

applications such as integrity checking and deduplication in file systems and etc. (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013)

The chart below is the speed comparison of various popular hash functions, taken from eBACS's "sandy" measurements. SHA-3 and BLAKE2 have no known security issues. SHA-1, MD5, SHA-256, and SHA-512 are susceptible to length-extension. SHA-1 and MD5 are vulnerable to collisions. MD5 is vulnerable to chosen-prefix collisions (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013).



BLAKE thus appears to be a good candidate for fast software hashing. BLAKE2 aims to provide the highest security level, be it in terms of classical notions as (second) preimage or collision resistance, or of theoretical notions as pseudo randomness or undifferentiability (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013).

Blake2 offers the following properties and capabilities (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013):

- Faster than MD5 on 64-bit Intel platforms.
- 32% less RAM required than BLAKE.

- Minimal padding, which is faster and simpler to implement.
- Direct support, with no overhead, of:
    - Parallelism for many-times faster hashing on multicore or SIMD (Single Instruction, Multiple Data) CPUs.
    - Tree hashing for incremental update or verification of large files.
    - Prefix-MAC for authentication that is simpler and faster than HMAC.
    - Personalization for defining a unique hash function for each application.

Consequently, it is evident that Blake2 is the best available hash function until now. We may change the hash if we find some flaw in Blake2 or if we find another hash function algorithm which offers higher assurance and performance. For more in-depth information about Blake2 capabilities, please visit the Blake2 official website.

2.2. Why Ed25519?

In previous section the importance and challenges of selecting a cryptographic hash function discussed carefully. In this a Public-key cryptography algorithm will be selected. There are plenty of theories that these algorithms are developed and created based on them such as: Discrete logarithm, Elliptic-curve cryptography, Non-commutative cryptography, RSA (Rivest–Shamir–Adleman) problem and Trapdoor function. Furthermore, for each mathematical algorithm which are already mentioned, there are numerous number of Public-key cryptography algorithms which are presented in the table below:

Public-key cryptography		
Cramer–Shoup	ECDSA	EdDSA
DH	EKE	Schnorr
DSA	ElGamal (signature scheme)	SPEKE
ECDH	MQV	SRP

A full list of available Public-key cryptographies' eBATS (ECRYPT Benchmarking of Asymmetric Systems) benchmarks available at:

- eBATS benchmarks for Ed25519.
- eBATS benchmarks for results of benchmarking of all Public-key signatures.
- eBATS benchmarks for full list of Public-key signatures

However, this is not the main reason for choosing Ed25519. In the rest of this section, the primary reasons of selecting Ed25519 among plenty of available Public-key signatures will be introduced. List of Ed25519 features is presented below as the major reasons for selecting Ed25519 (Bernstein, Duif, Lange, Schwabe, & Yang, 2011).

- **Fast single-signature verification** Ed25519 takes only 273364 cycles to verify a signature on Intel's widely deployed Nehalem/Westmere lines of CPUs. This performance measurement is for short messages.
- **Even faster batch verification** Ed25519 performs a batch of 64 separate signature verifications (verifying 64 signatures of 64 messages under 64 public keys) in only 8.55 million cycles, i.e., under 134000 cycles per signature. It fits easily into L1 cache, so contention between cores is negligible: a quad-core 2.4GHz Westmere verifies 71000 signatures per second, while keeping the maximum verification latency below 4 milliseconds.
- **Very fast signing** Ed25519 takes only 87548 cycles to sign a message.
- **Fast key generation** Key generation is almost as fast as signing. There is a slight penalty for key generation to obtain a secure random number from the operating system.
- **High security level** This system has a 2128 security target; The best attacks known actually cost more than 2140 bit operations on average, and degrade quadratically in success probability as the number of bit operations drops.
- **Foolproof session keys** Signatures are generated deterministically; key generation consumes new randomness but new signatures do not. This is not only a speed feature but also a security feature, directly

relevant to the recent collapse of the Sony PlayStation 3 security system.

- **Collision resilience** Hash-function collisions do not break this system. This adds a layer of defense against the possibility of weakness in the selected hash function.
- **No secret array indices** Ed25519 never reads or writes data from secret addresses in RAM; the pattern of addresses is completely predictable. The software is therefore immune to cache-timing attacks, hyper threading attacks, and other side-channel attacks that rely on leakage of addresses through the CPU cache.
- **No secret branch conditions** Ed25519 never performs conditional branches based on secret data; the pattern of jumps is completely predictable. The software is therefore immune to side-channel attacks that rely on leakage of information through the branch-prediction unit.
- **Small signatures** Signatures fit into 64 bytes. These signatures are actually compressed versions of longer signatures; the times for compression and decompression are included in the cycle counts reported above.
- **Small keys** Public keys consume only 32 bytes. The times for compression and decompression are again included.

Eventually, it is evident that Ed25519 is one of the best available Public-key signatures until now which is going to be used for digital signatures. We may change the Public-key signature if we find some flaw in Ed25519 or if we find another Public-key signature algorithm which offers higher assurance and performance. We picked Ed25519 algorithm because of the advantages this algorithm offering. For more in-depth information about Ed25519 features please visit Ed25519 official website.

2.3. Why Equihash?

There are four main consensus mechanisms, a consensus mechanism is fundamental problem in distributed computing and multi-agent systems is

to achieve overall system reliability in the presence of a number of faulty processes. These mechanisms listed as below:

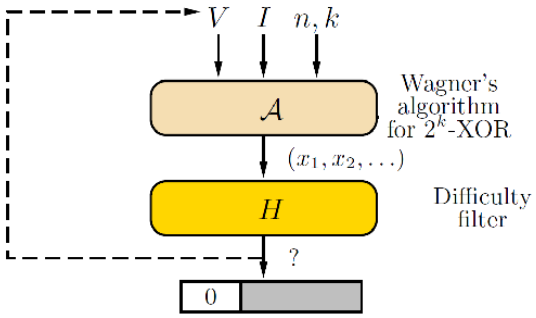
- Proof-of-Authority
- Proof-of-Space
- Proof-of-Stake
- Proof-of-Work

List of cryptocurrencies with Proof-of-Work consensus mechanism				
SHA-256-based	Equihash-based	Ethash-based	Scrypt-based	CryptoNote-based
Bitcoin	Zcash	Ethereum	Dogecoin	Bytecoin
Titcoin	Zcoin	Ubiq	PotCoin	Monero
Peercoin	Zclassic	KodakCoin	Litecoin	
NuBits	Bitcoin Gold	Ethereum Classic	Gulden	
Namecoin	Komodo			
Factom				
Bitcoin Cash				

In proof-of-work mining, miners are tasked with generating a short binary blob (called a nonce), which, when hashed, produces an output value less than a pre-specified target threshold. Due to the cryptographic nature of each currency's hash function, there is no way to reverse-engineer or back-compute a nonce that satisfies the target threshold limit. Instead, miners must "guess-and-check" hashes as fast as possible, and hope they're the first miner in the entire crypto-currency's network to find a valid nonce.

There are variety of algorithms based on this consensus mechanism such as Equihash and Ethash. Equihash is a memory-oriented Proof-of-Work algorithm developed by the University of Luxembourg's Interdisciplinary Centre for Security, Reliability and Trust (SnT). Zcash, Zcoin, Zclassic, Bitcoin Gold and Komodo all using Equihash while Ethereum, Ethereum Classic, KodakCoin and Ubiq are using Ethash. Equihash is a memory-oriented Proof-of-Work, which means how much mining one can do is mostly determined by how much RAM the computer has. The chart below shows how the Ethash works. The main advantage of Equihash over Ethash is simplicity in algorithm.

Equihash having the same speed on both CPU and GPU instead, Ethash has different speeds while running on CPU, GPU and FPGA. Finally, one can calculate the chance of reaching solution/solutions of Equihash by taking advantage of special probability. The diagram of Equihash algorithm is presented below from reference (Biryukov & Khovratovich, 2016):



Finally, we stress that we may change the Proof-of-Work, if we find some defect in Equihash or if we find another Proof-of-Work algorithm which offers higher performance, security and assurance. For more in-depth information about Equihash properties please, see (Biryukov & Khovratovich, 2016).

3. Introduction & Mathematical foundation

3.1. Blake2

In this section, the parameters and mathematical foundation of Blake2 (Blake2b and Blake2s) will be explained; adopted from reference (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013).

Parameter Blocks of Blake2b and Blake2s

Offset	0	1	2	3
0	Digest length	Key length	Fanout	Depth
4	Leaf length			
8 12	Node offset			
16	Node depth	Inner length	RFU	
20 24 28	RFU			
32 … 44	Salt			
48 … 60	Personalization			

BLAKE2b parameter block structure (offsets in bytes)

Offset	0	1	2	3
0	Digest length	Key length	Fanout	Depth
4	Leaf length			
8	Node offset			
12	Node offset (cont.)		Node depth	Inner length
16 20	Salt			
24 28	Personalization			

BLAKE2s parameter block structure (offsets in bytes)

- General parameters (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013):
  - Digest byte length (1 byte): an integer in [1, 64] for BLAKE2b, in [1, 32] for BLAKE2s
  - Key byte length (1 byte): an integer in [0, 64] for BLAKE2b, in [0, 32] for BLAKE2s (set to 0 if no key is used)
  - Salt (16 or 8 bytes): an arbitrary string of 16 bytes for BLAKE2b, and 8 bytes for BLAKE2s (set to all-NULL by default)
  - Personalization (16 or 8 bytes): an arbitrary string of 16 bytes for BLAKE2b, and 8 bytes for BLAKE2s (set to all-NULL by default)
- Tree hashing parameters (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013):
  - Fan-out an integer in [0, 255] (set to 0 if unlimited, and to 1 only in sequential mode)
  - Maximal depth (1 byte): an integer in [1, 255] (set to 255 if unlimited, and to 1 only in sequential mode)
  - Leaf maximal byte length (4 bytes): an integer in [0, 232 - 1], that is, up to 4 GiB (set to 0 if unlimited, or in sequential mode)
  - Node offset (8 or 6 bytes): an integer in [0, 264-1] for BLAKE2b, and in [0, 248-1] for BLAKE2s (set to 0 for the first, leftmost, leaf, or in sequential mode)
  - Node depth (1 byte): an integer in [0, 255] (set to 0 for the leaves, or in sequential mode)
  - Inner hash byte length (1 byte): an integer in [0, 64] for BLAKE2b, and in [0, 32] for BLAKE2s (set to 0 in sequential mode)

3.1.1. Blake2b

BLAKE2b supports data of any byte length $0 \leq l \leq 2^{128}$. Data is first padded to form a sequence of $N = \left\lceil \frac{l}{128} \right\rceil$ 16-word blocks $m_0, m_1, \ldots, m_{N-1}$ and then hashed by doing (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013):

$$h^0 \leftarrow IV \oplus P$$

$$\boldsymbol{for}\ i = 0, \ldots, N-1$$

$$h^{i+1} \leftarrow \boldsymbol{compress}\left(h^i, m^i, l^i\right)$$

$$\boldsymbol{return}\ h^N$$

Where l^i denotes the number of data bytes in $m^0, m^1, \ldots, m^{N-1}$ (that is, not counting any padding byte), P is the parameter block specified in **Parameter Blocks**, and IV is (as in BLAKE and SHA-512) the following 64-bit words (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013):

$$IV_0 = 6a09e667f3bcc908 \quad IV_1 = bb67ae8584caa73b$$
$$IV_2 = 3c6ef372fe94f82b \quad IV_3 = a54ff53a5f1d36fa$$
$$IV_4 = 510e527fade682d1 \quad IV_5 = 9b05688c2b3e6c1f$$
$$IV_6 = 1f83d9abfb41bd6b \quad IV_7 = 5be0cd19137e2179$$

The compression function **compress** takes as input (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013):

- A 64-byte chain value $h = h_0, \ldots, h_7$
- A 128-byte message block $m = m_0, \ldots, m_{15}$
- A counter $t = t_0, t_1$, and finalization flags f_0, f_1

First, compress initializes a 16-word internal state $v_0, \ldots, v_{15}$ that is (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013):

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} = \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ IV_0 & IV_1 & IV_2 & IV_3 \\ t_0 \oplus IV_4 & t_1 \oplus IV_5 & f_0 \oplus IV_6 & f_1 \oplus IV_7 \end{pmatrix}$$

Where f_0 and f_1 are the finalization flags. The internal state v is then transformed through a sequence of 12 rounds, where a round does (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013):

$$G_0(v_0, v_4, v_8, v_{12}) \quad G_1(v_1, v_5, v_9, v_{13}) \quad G_2(v_2, v_6, v_{10}, v_{14}) \quad G_3(v_3, v_7, v_{11}, v_{15})$$
$$G_4(v_0, v_5, v_{10}, v_{15}) \quad G_5(v_1, v_6, v_{11}, v_{12}) \quad G_6(v_2, v_7, v_8, v_{13}) \quad G_7(v_3, v_4, v_9, v_{14})$$

That is, a round applies a G function to each of the columns in parallel, and then to all of the diagonals in parallel. The G function of BLAKE2b uses the constants in the table below which is permutations of $\mathrm{f}\{0,\ldots,15\}$ used by the BLAKE2 functions (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013).

σ_0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
σ_1	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
σ_2	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
σ_3	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
σ_4	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
σ_5	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
σ_6	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
σ_7	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
σ_8	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
σ_9	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

After the 12 rounds, the new chain value $h'_0,\ldots,h'_7$ is defined as (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013):

$$h'_0 = h_0 \oplus v_0 \oplus v_8$$

$$h'_1 = h_1 \oplus v_1 \oplus v_9$$

$$h'_2 = h_2 \oplus v_2 \oplus v_{10}$$

$$h'_3 = h_3 \oplus v_3 \oplus v_{11}$$

$$h'_4 = h_4 \oplus v_4 \oplus v_{12}$$

$$h'_5 = h_5 \oplus v_5 \oplus v_{13}$$

$$h'_6 = h_6 \oplus v_6 \oplus v_{14}$$

$$h'_7 = h_7 \oplus v_7 \oplus v_{15}$$

Note the absence of the salt, compared to BLAKE.

3.1.2. Blake2s

BLAKE2s supports data of any byte length $0 \leq \mathrm{l} \leq 2^{64}$. It works similarly to BLAKE2b, but on 32-bit words instead of 64-bit words (the byte length of a

chaining value, a message block, a counter or finalization flag are thus divided by two). BLAKE2s uses the following IV:

$$IV_0 = 6a09e667 \qquad IV_1 = bb67ae85$$
$$IV_2 = 3c6ef372 \qquad IV_3 = a54ff53a$$
$$IV_4 = 510e527f \qquad IV_5 = 9b05688c$$
$$IV_6 = 1f83d9ab \qquad IV_7 = 5be0cd19$$

BLAKE2s does 10 rounds, and uses the G function. For more in-depth mathematical information about Blake2b and Blake2s, please visit (Aumasson, Neves, Wilcox-O'Hearn, & Winnerlein, 2013).

3.2. Ed25519

In this section the mathematical foundation of Ed25519 and the governing algorithm of Ed25519 will be explained. This section adopted from references (Langley, Google, Hamburg, Rambus Cryptography Research, & Turner, 2016), (Josefsson & Liusvaara, 2017).

Parameters

1. An odd prime power p. EdDSA uses an elliptic curve over the finite field $GF(p)$.
2. An integer b with $2(b-1) > p$. EdDSA public keys have exactly b bits, and EdDSA signatures have exactly 2b bits. b is recommended to be a multiple of 8, so public key and signature lengths are an integral number of octets.
3. A $(b-1)$-bit encoding of elements of the finite field $GF(p)$.
4. A cryptographic hash function H producing $2b$ -bit output. Conservative hash functions (i.e., hash functions where it is infeasible to create collisions) are recommended and do not have much impact on the total cost of EdDSA.
5. An integer c that is 2 or 3. Secret EdDSA scalars are multiples of 2c. The integer c is the base-2 logarithm of the so-called cofactor.

6. An integer n with $c \leq n < b$. Secret EdDSA scalars have exactly $n + 1$ bits, with the top bit (the $2n$ position) always set and the bottom c bits always cleared.
7. A non-square element d of $GF(p)$. The usual recommendation is to take it as the value nearest to zero that gives an acceptable curve.
8. A non-zero square element of $GF(p)$. The usual recommendation for best performance is $a = -1$ if $p \bmod 4 = 1$, and $a = 1$ if $p \bmod 4 = 3$.
9. An element $B \neq (0,1)$ of the set $E = \{(x, y) \in GF(p) \times GF(p) \mid\mid ax^2 + y^2 = 1 + dx^2y^2\}$.
10. An odd prime L such that $[L]B = 0$ and $2x^cL = \#E$. The number $\#E$ (the number of points on the curve) is part of the standard data provided for an elliptic curve E, or it can be computed as cofactor * order.
11. A "prehash" function PH. PureEdDSA means EdDSA where PH is the identity function, i.e., $PH(M) = M$. HashEdDSA means EdDSA where PH generates a short output, no matter how long the message is; for example, $PH(M) = Hash(M)$.
12. Points on the curve form a group under addition, $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$, with the formulas:

$$x_3 = \frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2} \quad y_3 = \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2}$$

The neutral element in the group is $(0,1)$. Unlike many other curves used for cryptographic applications, these formulas are "complete"; they are valid for all points on the curve, with no exceptions. In particular, the denominators are non-zero for all input points. There are more efficient formulas, which are still complete, that use homogeneous coordinates to avoid the expensive modulo p inversions. Ed25519 is EdDSA instantiated with:

p $2^{255} - 19$

b 256

Encoding of GF(p) 255-bit little-endian encoding of $\{0, 1, \ldots, p - 1\}$

c $\log_2(cofactor\ of\ edwards25519)$

Cofactor of edwards25519 = 8

n 254

d
37095705934669439343138083508754565189542113879843219016388785533085940283555

a -1

B $(X(P), Y(P))$ of edwards25519

$X(P)$
15112221349535400772501151409588531511454012693041857206046113283949847762202

$Y(P)$
46316835694926478169428394003475163141307993866256225615783033603165251855960

L order $2^{252} + 0x14def9dea2f79cd65812631a5cf5d3ed$

Modular Arithmetic

For advice on how to implement arithmetic modulo $p = 2^{255} - 19$ efficiently and securely, see Curve25519 (Josefsson & Liusvaara, 2017). For inversion modulo p, it is recommended to use the identity $x^{-1} = x^{p-2}\ (mod\ p)$. Inverting zero should never happen, as it would require invalid input, which would have been detected before, or would be a calculation error. For point decoding or "decompression", square roots modulo p are needed. They can be computed using the Tonelli-Shanks algorithm or the special case for p = 5 (mod 8). To find a square root of a, first compute the candidate root $x = a^{\frac{p+3}{8}}\ (mod\ p)$. Then there are three cases:

1. $x^2 = a\ (mod\ p)$. Then x is a square root.
2. $x^2 = -a\ (mod\ p)$. Then $2^{\frac{p-1}{4}}x$ is a square root.

3. a is not a square modulo p.

Encoding

All values are coded as octet strings, and integers are coded using little-endian convention, i.e., a 32-octet string $h = h[0], \ldots, h[31]$ represents the integer $h[0] + 2^8 * h[1] + \ldots + 2^{248} * h[31]$. A curve point (x, y), with coordinates in the range $0 \leq x, y < p$, is coded as follows. First, encode the y-coordinate as a little-endian string of 32 octets. The most significant bit of the final octet is always zero. To form the encoding of the point, copy the least significant bit of the x-coordinate to the most significant bit of the final octet.

Decoding

Decoding a point, given as a 32-octet string, is a little more complicated.

1. First, interpret the string as an integer in little-endian representation. Bit 255 of this number is the least significant bit of the x-coordinate and denote this value x_0. The y-coordinate is recovered simply by clearing this bit. If the resulting value is $\geq p$, decoding fails.
2. To recover the x-coordinate, the curve equation implies $x^2 = \frac{y^2-1}{dy^2+1} \ (mod\ p)$. The denominator is always non-zero mod p. Let $u = y^2 - 1$ and $v = dy^2 + 1$. To compute the square root of $\left(\frac{u}{v}\right)$, the first step is to compute the candidate root $x = \left(\frac{u}{v}\right)^{\frac{p+3}{8}}$. This can be done with the following trick, using a single modular powering for both the inversion of v and the square root:

$$x = \left(\frac{u}{v}\right)^{\frac{p+3}{8}} = uv^3(uv^7)^{\frac{p-5}{8}} \ (mod\ p)$$

3. Again, there are three cases:

3.1. If $vx^2 = u \ (mod\ p)$, x is a square root.

3.2. If $vx^2 = -u \ (mod\ p)$, set $x \leftarrow x2^{\frac{p-1}{4}}$, which is a square root.
3.3. Otherwise, no square root exists for modulo p, and decoding fails.

4. Finally, use the x_0 bit to select the right square root. If $x = 0$, and $x_0 = 1$, decoding fails. Otherwise, if $x_0 \neq x \ mod\ 2$, set $x \leftarrow p - x$. Return the decoded point (x, y).

Point Addition

For point addition, the following method is recommended. A point (x, y) is represented in extended homogeneous coordinates (X, Y, Z, T), with $x = \frac{X}{Z}$, $y = \frac{Y}{Z}$, $xy = \frac{T}{Z}$. The neutral point is $(0,1)$, or equivalently in extended homogeneous coordinates $(0, Z, Z, 0)$ for any non-zero Z.

The following formulas for adding two points, $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$, on twisted Edwards curves with $a = -1$, square a, and non-square d are described in Section 3.1 of Edwards-revisited and in EFD-TWISTED-ADD. They are complete, i.e., they work for any pair of valid input points.

$$A = (Y_1 - X_1)(Y_2 - X_2)$$
$$B = (Y_1 + X_1)(Y_2 + X_2)$$
$$C = T_1 2dT_2$$
$$D = Z_1 2Z_2$$
$$E = B - A$$
$$F = D - C$$
$$G = D + C$$
$$H = B + A$$
$$X_3 = E * F$$
$$Y_3 = G * H$$
$$T_3 = E * H$$

$$Z_3 = F * G$$

For point doubling, $(x_3, y_3) = (x_1, y_1) + (x_1, y_1)$, one could just substitute equal points in the above (because of completeness, such substitution is valid) and observe that four multiplications turn into squares. However, using the formulas described in Section 3.2 of Edwards-revisited and in EFD-TWISTED-ADD saves a few smaller operations.

$$A = X_1^2$$

$$B = Y_1^2$$

$$C = 2Z_1^2$$

$$H = A + B$$

$$E = H - (X_1 + Y_1)^2$$

$$G = A - B$$

$$F = C + G$$

$$X_3 = E * F$$

$$Y_3 = G * H$$

$$T_3 = E * H$$

$$Z_3 = F * G$$

Key Generation

The private key is 32 octets (256 bits, corresponding to b) of cryptographically secure random data. See RFC4086 for a discussion about randomness. The 32-byte public key is generated by the following steps.

1. Hash the 32-byte private key, storing the digest in a 64-octet large buffer, denoted h. Only the lower 32 bytes are used for generating the public key.

2. Run the buffer: The lowest three bits of the first octet are cleared, s the highest bit of the last octet is cleared, and the second highest bit of the last octet is set.
3. Interpret the buffer as the little-endian integer, forming a secret scalar s. Perform a fixed-base scalar multiplication $[s]B$.
4. The public key A is the encoding of the point $[s]B$. First, encode the y-coordinate (in the range $0 \leq y < p$) as a little- endian string of 32 octets. The most significant bit of the final octet is always zero. To form the encoding of the point $[s]B$, copy the least significant bit of the x coordinate to the most significant bit of the final octet. The result is the public key.

Sign

The input to the signing procedure is the private key, a 32-octet string, and a message M of arbitrary size.

1. Hash the private key, 32 octets. Let h denote the resulting digest. Construct the secret scalar s from the first half of the digest, and the corresponding public key A, as described in the previous section. Let prefix denote the second half of the hash digest, $h[32], \ldots, h[63]$.
2. Compute $Hash(M)$, where M is the message to be signed. Interpret the 64-octet digest as a little- endian integer r.
3. Compute the point $[r]B$. For efficiency, do this by first reducing r modulo L, the group order of B. Let the string R be the encoding of this point.
4. Compute $Hash(\ R \ || \ A)$, and interpret the 64-octet digest as a little-endian integer k.
5. Compute $S = (r + k \times s)$ mod L. For efficiency, again reduce k modulo L first.
6. Form the signature of the concatenation of R (32 octets) and the little-endian encoding of S (32 octets; the three most significant bits of the final octet are always zero).

Verify

1. To verify a signature on a message M using public key A, with F being 0 for Ed25519 and if Ed25519 is being used, C being the context, first split the signature into two 32-octet halves. Decode the first half as a point R, and the second half as an integer S, in the range $0 \leq s < L$. Decode the public key A as point A'. If any of the decodings fail (including S being out of range), the signature is invalid.
2. Compute $Hash(R \,||\, A)$, and interpret the 64-octet digest as a little-endian integer k.
3. Check the group equation $[8][S]B = [8]R + [8][k]A'$. It's sufficient, but not required, to instead check $[S]B = R + [k]A'$.

3.3. Equihash

Wagner exposes the generalized birthday problem and the algorithm for it (Wagner, 2002). The generalized birthday problem for one list is formulated as follows: given list L of n-bit strings $\{X_i\}$, find distinct $\{X_{i_j}\}$ such that (Biryukov & Khovratovich, 2016):

$$X_{i_1} \oplus X_{i_2} \oplus \ldots \oplus X_{i_3} = 0$$

Wagner considers the setting where $\{X_i\}$ are outputs of some (non-keyed) PRNG, e.g. a hash function H in the counter mode. Thus Wagner had to find $\{i_j\}$ such that (Biryukov & Khovratovich, 2016):

$$H(i_1) \oplus H(i_2) \oplus \ldots \oplus H(i_3) = 0$$

For $k = 1$ this problem is the collision search, and can be solved trivially by sorting in $2^{\frac{n}{2}}$ time and space complexity if $|L| > 2^{\frac{n}{2}}$. However, for $k > 1$ and smaller lists the problem is harder. For instance, from the information-theoretic point of view a solution for $k = 2$ in a list of size $2^{\frac{n}{4}}$ is expected, but no algorithm faster than $2^{\frac{n}{3}}$ operations is known (Biryukov & Khovratovich, 2016).

Wagner demonstrated an algorithm for $k > 1$ and the lists are large enough to contain numerous solutions. It has time and space complexity of $O\left(2^{\frac{n}{k+1}}\right)$ for lists of the same size. Wagner's algorithm generalizes easily to some operations other than XOR (e.g., to the modular addition). Also note that for $k \log_2 n$ a XOR solution can be found by the much faster Gaussian elimination with complexity of $O(2^k)$ string operations. The table below adopted from (Wagner, 2002) with some modifications explains the basic Wagner algorithm for the generalized birthday problem.

Basic Wagner's algorithm for the generalized birthday problem

Input: list L of N n-bit strings ($N \ll 2^n$).

(1) Enumerate the list as $\{X_1, \dots, X_N\}$ and store pairs (X_j, j) in a table.

(2) Sort the table by X_j. Then find all unordered pairs (i, j) such that X_i collides with X_j on the first $\frac{n}{k+1}$ bits. Store all tuples $(X_{i,j} = X_i \oplus X_j, i, j)$ in the table.

(3) Repeat the previous step to find collisions in $X_{i,j}$ on the next $\frac{n}{k+1}$ bits and store the resulting tuples $(X_{i,j,k,l}, i, j, k, l)$ in the table.

(4) [...] Repeat the previous step for the next $\frac{n}{k+1}$ bits, and so on until only $\frac{2n}{k+1}$ bits are non-zero.

(5) $[k+1]$ At the last step, find a collision on the last $\frac{2n}{k+1}$ bits. This gives a solution to the original problem.

Output: list $\{i_j\}$ conforming to $H(i_1) \oplus H(i_2) \oplus \dots \oplus H\left(i_{2^k}\right) = 0$.

Assume that sorting $l = O(N)$ elements is computationally equivalent to l calls to the hash function H. Let a single call to H be the time unit (Biryukov & Khovratovich, 2016).

Proposition 1. For $N = 2^{\frac{n}{k+1}} + 1$ and $k^2 < n$ Algorithm above produces two solutions (on average) using $\frac{(2^{k-1}+n)N}{8}$ bytes of memory in time $(k+1)N$ (Biryukov & Khovratovich, 2016).

Proof. Suppose $N = 2^{\frac{n}{k+1}} + 1$ tuples stored at the first step. Then after collision search it is expected to have: (Biryukov & Khovratovich, 2016):

$$\frac{\frac{N(N-1)}{2}}{\frac{N}{2}} = N - 1$$

Entries for the second table, then $N - 3$ entries for the third table, and so on. Before the last (k-th) collision search it is expected to $N - 2^{k-1} + 1$ is approximatly $N = 2^{\frac{n}{k+1}} + 1$ entries, thus on average two solutions after the last step were obtained. The computational complexity is dominated by the complexity of list generation (N hash calls) and subsequent k sorting of N elements. Therefore, the total computational complexity is equivalent to (Biryukov & Khovratovich, 2016):

$$(k+1)N = (k+1)N = 2^{\frac{n}{k+1}} + 1$$

Hash function calls. This ends the proof. Biryukov et. al have not computed the variance of the number of solutions, but their experiments demonstrate that the actual number of solutions at each step is very close (within 10%) to the expected number. If larger lists are used, the table will grow in size over the steps. The list size is exactly taken so that the expected number of solutions is small and the table size does not change much (Biryukov & Khovratovich, 2016).

The generalized birthday problem in its basic form lacks some necessary properties as a proof-of-work. The reason is that Wagner's algorithm can be iterated to produce multiple solutions by selecting other sets of colliding bits or using more sophisticated techniques. If more memory is available, these solutions can be produced at much lower amortized cost (Proposition 3). Since this property violates the non-amortization requirement for the PoW (see [6]), it was suggested to modify the problem so that only two solutions can be produced on average (Biryukov & Khovratovich, 2016).

This modification is inspired by the fact that a solution found by Wagner's algorithm carries its footprint. Namely, the intermediate $2^l - XORs$ have

leading $\frac{nl}{k+1}$ bits, for example $X_{i_4} \oplus X_{i_5} \oplus X_{i_6} \oplus X_{i_7}$ collide on certain $\frac{2n}{k+1}$ bits. Therefore, if the positions are pre-fixed, where $2^l - XORs$ have zero bits, the user will be bind to a particular algorithm flow. Moreover, it has been proven that the total number of possible solutions that conform to these restrictions is only 2 on average, so that the problem becomes amortization-free for given input list L. Only duplicate solutions should be taken care of which appear by swapping $2^l - XORs$ within the $2^l - XORs$, for any l. It is simply required that every $2^l - XORs$ is ordered as a pair, e.g. with lexicographic order. It should be considered that a certain order is a prerequisite as otherwise duplicate solutions (produced by swaps in pairs, swaps of pairs, etc.) would be accepted. With this modification the Gaussian elimination algorithm does not apply any longer, so larger k can be adopted with no apparent drop in complexity (Biryukov & Khovratovich, 2016).

Proposition 2. Optimized Wagner's algorithm below which adopted from (Biryukov & Khovratovich, 2016) with some changes:

Optimized Wagner's algorithm for the generalized birthday problem

Input: list L of N n-bit strings.

(1) Enumerate the list as $\{X_1, \dots, X_N\}$ and store pairs (j) in a table.

(2) Sort the table by X_j, computing it on-the-fly. Then find all unordered pairs (i, j) such that X_i collides with X_j on the first $\frac{n}{k+1}$ bits. Store all such pairs (i, j) in the table.

(3) Repeat the previous step to find collisions in $X_{i,j}$ (again recomputing on-the-fly) on the next $\frac{n}{k+1}$ bits and store the resulting tuples (i, j, k, l) in the table.

(4) [...] Repeat the previous step for the next $\frac{n}{k+1}$ bits, and so on. When indices trimmed to 8 bits plus the length $X_{i,j,\dots}$ becomes smaller than the full index tuple, switch to trimming indices.

(5) $[k + 1]$ At the last step, find a collision on the last $\frac{2n}{k+1}$ bits. This gives a solution to the original problem.

Output: list $\{i_j\}$ conforming to $H(i_1) \oplus H(i_2) \oplus \dots \oplus H\left(i_{2^k}\right) = 0$.

For $N = 2^{\frac{n}{k+1}} + 1$ runs in $M(n,k) = 2^{\frac{n}{k+1}}\left(2^2 + \left(\frac{n}{2(k+1)}\right)\right)$ bytes of memory and $T(n,k) = k2^{\left(\frac{n}{k+1}\right)+2}$ time (Biryukov & Khovratovich, 2016).

Proposition 3. Using $qM(n,k)$ memory, a user can find $2q^{k+1}$ solutions with cost $qT(n,k)$, so that the amortized cost drops by q^{k+1} (Biryukov & Khovratovich, 2016).

Proposition 4 Using $M(n,k) = q$ memory, a user can find 2 solutions in time$C1(q)T(n,k)$, where [6]:

$$C_1(1) \approx \frac{3q^{\frac{k-1}{2}} + k}{k+1}$$

Therefore, Wagner's algorithm for finding $2^k - XOR$ has a tradeoff of steepness $\frac{k-1}{2}$. At the cost of increasing the solution length, the penalty for memory-reducing users can increase (Biryukov & Khovratovich, 2016).

Proposition 5. Using constant memory, a user can find one algorithm-bound solution in time (Biryukov & Khovratovich, 2016).

$$2^{\frac{n}{2}+2k+\frac{n}{k+1}}$$

Proposition 6. Using $\frac{M(n,k)}{q}$ memory, a user can find 2 algorithm-bound solutions in time $C_2(q)T(n,k)$, where (Biryukov & Khovratovich, 2016):

$$C_2(q) \approx 2^k q^{\frac{k}{2}} k^{\frac{k}{2}-1}$$

Therefore, the algorithm-bound proof-of-work has higher steepness (k/2), and the constant is larger. Thus far it have been equalized the time and computational complexity, whereas an ASIC-equipped user or one with a multi-core cluster would be motivated to parallelize the computation if this reduces the AT cost [(Biryukov & Khovratovich, 2016).

Proposition 7. With $p \ll T(n,k)$ processors and $M(n,k)$ shared memory a user can find 2 algorithm-bound solutions in time.

$$\frac{T(n,k)}{p}(1-\log_N p)$$

Additionally, the memory bandwidth grows by the factor of p. For fixed memory size, memory chips with bandwidth significantly higher than that of typical desktop memory (such as DDR3) are rare. Assuming that a prover does not have access to memory with bandwidth higher than certain Bwmax (maximum bandwidth), one can efficiently bound the time-memory (and thus the time-area) product for such implementations (Biryukov & Khovratovich, 2016).

To generate an instance for the proof protocol, a verifier selects a cryptographic hash function H and integers n, k, d which determine time and memory requirements as follows (Biryukov & Khovratovich, 2016):

- $Memory\ M\ is\ 2^{\frac{n}{k+1}} + k$
- $Time\ T\ is\ (k+1)2^{\left(\frac{n}{k+1}\right)+d}\ calls\ to\ the\ hash\ function\ H.$
- Solution size is $2^k\left(\left(\frac{n}{k+1}\right)\ +\ 1\right)$ + 160 bits.
- Verification cost is 2^k hashes and XORs.

Then by selecting a seed I (which may be a hash of transactions, block chaining variable, etc.) and the prover will be tasked with finding 160-bit nonce V and $\left(\left(\frac{n}{k}+1\right)+1\right)-bit\ \ x_1, x_2, \ldots, x_{2^k}$ such that (Biryukov & Khovratovich, 2016):

Generalized Birthday - Condition

$$H\left(I\middle||V|\middle|x_1\right)\ \oplus\ H\left(I\middle||V|\middle|x_2\right)\ \oplus\ \ldots\ \oplus\ H\left(I\middle||V|\middle|x_{2^k}\right)\ =\ 0$$

Difficulty - Condition

$$H(I||V||x_1||x_2||\ldots||x_{2^k})\ has\ d\ leading\ zeroes$$

Algorithm Binding - Condition

$$H\left(I\middle||V|\middle|x_{w2^l+1}\right)\ \oplus\ \ldots\ \oplus\ H\left(I\middle||V|\middle|xw2^l+2^{l-1}\right)$$

$$has\ \frac{nl}{k+1}\ \ leading\ zeroes\ for\ all\ w, l$$

$(xw2^l + 1||xw2^l + 2||\ldots||xw2^l + 2^{l-1}) < (xw2^l + 2^{l-1} + 1||xw2^l + 2^{l-1} + 2||\ldots||xw2^l + 2^l)$ lexicographically

$$For\ all\ l\ \in \{1, \ldots, k-1\}\ and\ For\ all\ w \in \{0, \ldots, 2^{k-l} - 1\}$$

Here the order is lexicographical. A prover is supposed to run Wagner's algorithm and then H. For more information about the Equihash and proof of propositions please, see reference (Biryukov & Khovratovich, 2016).

4. How to get started with it?

In this section the packages related to each algorithm, parameters required to use them and the test results will be presented. Each subsection introducing the test vectors or testing parameters.

4.1. Let's get started with Blake2-Node.js-Addon

In this section the performance of two sets of packages created for Blake2 will be compared. These packages which are forked from other projects; one is the Pure JavaScript implementation of Blake2 and the other one is Node.js C/C++ Addon created for Blake2. The difference is crystal clear in the name, but the main differences are going to be shown in the subsections below. This package is in accordance with Blake2 RCF 7693 Standard, the final results also compared as well; more test vectors can be accessed there as well.

4.1.1. Blake2 Pure JavaScript-Node-Addon

This packages which is forked and created from other projects, is tested for generating hash using Blake2b and Blake2s. A JavaScript code is created to test the performance of the package which is available at Appendix A: Blake2b-PureJava-test. The result of hashing “abc” as the input string presented below:

ba80a53f981c4d0d6a2797b69f12f6e94c212f14685ac4b74b12bb6fdbffa2d17d87c5392aab792dc252d5de4533cc9518d38aa8dbf1925ab92386edd4009923

The picture below is the test result of “**abc”** published in RCF 7693:

```
BLAKE2b-512("abc") = BA 80 A5 3F 98 1C 4D 0D 6A 27 97 B6 9F 12 F6 E9
                     4C 21 2F 14 68 5A C4 B7 4B 12 BB 6F DB FF A2 D1
                     7D 87 C5 39 2A AB 79 2D C2 52 D5 DE 45 33 CC 95
                     18 D3 8A A8 DB F1 92 5A B9 23 86 ED D4 00 99 23
```

The result of calculating the number of hashes Blake2b - Pure JavaScript can generate: **219,743** Blake2b with the digest length of 512 bits in 5 second.

Performing the same test for the Blake2s, with the input vector of "**abc**" and by developing a JavaScript code available at Appendix B: Blake2s-PureJava-test.

508c5e8c327c14e2e1a72ba34eeb452f37458b209ed63a294d999b4c86675982

The picture below is the test result of "**abc**" published in RCF 7693:

```
BLAKE2s-256("abc") = 50 8C 5E 8C 32 7C 14 E2 E1 A7 2B A3 4E EB 45 2F
                     37 45 8B 20 9E D6 3A 29 4D 99 9B 4C 86 67 59 82
```

The result of calculating the number of hashes Blake2s - Pure JavaScript can generate: **450,680** Blake2s with the digest length of 256 bits in 5 second.

4.1.2. Blake2 C/C++ Node-Addon

This package is forked from other projects and developed to create a C/C++ Node.js Addon. The difference is JavaScript packages are slower as JavaScript is a high level programming language. The main platform in TrustNote is Node.js; therefor to have the best performance while using the basis platform, it will be required to create a C/C++ Addon. Consequently, a JavaScript code will be used as the interface and the calculation will be done in C/C++ core which this favors the performance. The performance of the same functions with the same input in this package will be presented.

Same as previous section the test is to create hash using Blake2b and Blake2s with the digest length of 512 and 256 respectively. A JavaScript code is created to test the performance of the package which is available at Appendix C: Blake2b-C/C++ Node.js-Addon-test.

The result of hashing "**abc**" as the input string presented below:

ba80a53f981c4d0d6a2797b69f12f6e94c212f14685ac4b74b12bb6fdbffa2d17d87c5392aab792dc252d5de4533cc9518d38aa8dbf1925ab92386edd4009923

The picture below is the test result of "**abc**" published in RCF 7693:

```
BLAKE2b-512("abc") = BA 80 A5 3F 98 1C 4D 0D 6A 27 97 B6 9F 12 F6 E9
                     4C 21 2F 14 68 5A C4 B7 4B 12 BB 6F DB FF A2 D1
                     7D 87 C5 39 2A AB 79 2D C2 52 D5 DE 45 33 CC 95
                     18 D3 8A A8 DB F1 92 5A B9 23 86 ED D4 00 99 23
```

The result of calculating the number of hashes Blake2b - Pure JavaScript can generate: **6,605,304** Blake2b with the digest length of 512 bits in 5 second.

Performing the same test for the Blake2s, with the input vector of "**abc**" and by developing a JavaScript code available at Appendix D: Blake2s-C/C++ Node.js-Addon-test.

508c5e8c327c14e2e1a72ba34eeb452f37458b209ed63a294d999b4c86675982

The picture below is the test result of "**abc**" published in RCF 7693:

```
BLAKE2s-256("abc") = 50 8C 5E 8C 32 7C 14 E2 E1 A7 2B A3 4E EB 45 2F
                     37 45 8B 20 9E D6 3A 29 4D 99 9B 4C 86 67 59 82
```

The result of calculating the number of hashes Blake2s - Pure JavaScript can generate: **7,128,410** Blake2s with the digest length of 256 bits in 5 second.

Therefore, it is crystal clear that C/C++ Addon generates **6,605,304** hashes for Blake2b and **7,128,410** hashes for Blake2s with maximum digest length while Pure JavaScript generates only **219,743** hashes for Blake2b and **450,680** hashes for Blake2s.

4.2. Let's get started with Ed25519-Node-Addon

The previous chapter showed that it's a waste of time to use Pure JavaScript packages for the project. Therefore, only the C/C++ Addon performance tested and presented for this package. The JavaScript code developed to test this package which is forked from other projects, is available at Appendix E:

Ed25519-C/C++ Node.js-Addon-test. During 5 seconds, this package generates, signs and verifies 4,198 keys. This package presents an amazing performance.

4.3. Let's get started with Equihash

This package is forked from other projects. This package will be performed in C/C++ platform. This package tested with different input and, the outcomes are presented in the table below:

N (bits)	K	Seed	Time (ms)	Difficulty	Solution Size (KB)	Number of solutions found
100	4	5	5,155	1	81,920	16
100	4	20	15,281	1	81,920	16
108	5	5	2,843	1	25,600	32
108	5	20	8,499	1	25,600	32
110	4	20	21,967	1	327,680	16
110	4	5	22,062	1	327,680	16
126	5	5	49,077	1	204,800	32
126	5	20	36,797	1	204,800	32

5. Discussion

This document aimed to introduce the most important subjects which directly effecting the security and performance of the final product. Eventually, TrustNote will be manipulating Blake2 as its hash function, Ed25519 as its public key generator for signing contracts and Equihash for verification. As mentioned before, this is a work-in-progress so if we find any newer and better algorithm which guarantees security of the users while offering a higher performance, we will replace it with our current packages.

6. References

Aumasson , J.-P., Neves , S., Wilcox-O'Hearn, Z., & Winnerlein , C. (2017, 02 22). *BLAKE2 — fast secure hashing*. Retrieved from BLAKE2: https://Blake2.net/https://Blake2.net/

Aumasson, J.-P., Neves, S., Wilcox-O'Hearn, Z., & Winnerlein, C. (2013). BLAKE2: simpler, smaller, fast as MD5.

Bernstein, D., Duif, N., Lange, T., Schwabe, P., & Yang, B.-Y. (2011). Ed25519: high-speed high-security signatures.

Biryukov, A., & Khovratovich, D. (2016). Equihash: asymmetric proof-of-work based on the Generalized Birthday problem. *LEDGER*.

Chang, S.-j., Perlner, R., Burr, W., Turan, M., Kelsey, J., Paul, S., & Bassham, L. (2012). *Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition.* National Institute for Standards and Technology.

Josefsson, S., & Liusvaara, I. (2017). *Edwards-Curve Digital Signature Algorithm (EdDSA).* Internet Research Task Force.

Langley, A., Google, Hamburg, M., Rambus Cryptography Research, & Turner, S. (2016). *Edwards-Curve Digital Signature Algorithm.*

Wagner, D. (2002). A Generalized Birthday Problem. *Lecture Notes in Computer Science 2442*, 288–303.

Appendix

A: Blake2b-PureJava-test

```
'use strict';
var startTime = new Date().getTime();
var count = 0;
var TD = 0;
var test = require('tape')
var Blake2b = require('./Blake2b')
var util = require('./util')
var fs = require('fs')
var Blake2bHex = Blake2b.Blake2bHex
var hash = Blake2bHex('abc');
console.log('Blake2b result for input array "abc"');
console.log(hash.toString('hex'));
var endTime = new Date().getTime();
TD = endTime  - startTime;
while ((endTime  - startTime) < 5000){
    var hash = Blake2bHex('abc');
    var endTime = new Date().getTime();
    count++;}
console.log('Number of hashes generated during 5 seconds:');
console.log(count)
```

B: Blake2s-PureJava-test

```
'use strict';
var startTime = new Date().getTime();
var count = 0;
var TD = 0;
var test = require('tape')
var toHex = require('./util').toHex
var util = require('./util')
var b2s = require('./Blake2s')
var Blake2s = b2s.Blake2s
var Blake2sHex = b2s.Blake2sHex
var Blake2sInit = b2s.Blake2sInit
var Blake2sUpdate = b2s.Blake2sUpdate
var Blake2sFinal = b2s.Blake2sFinal
var hash = Blake2sHex('abc');
console.log('Blake2s result for input array "abc"');
console.log(hash.toString('hex'));
var endTime = new Date().getTime();
TD = endTime  - startTime;
while ((endTime  - startTime) < 5000) {
    var hash = Blake2sHex('abc');
    var endTime = new Date().getTime();
    count++;}
console.log('Number of hashes generated during 5 seconds:');
console.log(count)
```

C: Blake2b-C/C++ Node.js-Addon-test

```
'use strict';
var startTime = new Date().getTime();
var count = 0;
var TD = 0;
var
demand = require('must'),
fs        = require('fs'),
path      = require('path'),
Blake2    = require('./index')
;
var startTime = new Date().getTime();
var count = 0;
var TD = 0;
var assert = require('assert');
var buf = new Buffer('abc');
var hash = Blake2.sumBuffer(buf, Blake2.ALGORITHMS.B);
assert(hash instanceof Buffer);
console.log('Blake2b:');
console.log(hash.toString('hex'));
var endTime = new Date().getTime();
TD = endTime  - startTime;
while ((endTime  - startTime) < 5000) {
    var hash = Blake2.sumBuffer(buf, Blake2.ALGORITHMS.B);
    assert(hash instanceof Buffer);
    var endTime = new Date().getTime();
    count++;}
console.log('Number of hashes generated during 5 seconds:');
console.log(count)
```

D: Blake2s-C/C++ Node.js-Addon-test

```
'use strict';
var startTime = new Date().getTime();
var count = 0;
var TD = 0;
var assert = require('assert');
var buf = new Buffer('abc');
var hash2 = Blake2.sumBuffer(buf, Blake2.ALGORITHMS.S);
assert(hash2 instanceof Buffer);
console.log('Blake2s:');
console.log(hash2.toString('hex'));
var endTime = new Date().getTime();
TD = endTime  - startTime;
while ((endTime  - startTime) < 5000) {
    var hash2 = Blake2.sumBuffer(buf, Blake2.ALGORITHMS.S);
    assert(hash2 instanceof Buffer);
    var endTime = new Date().getTime();
    count++;}
console.log('Number of hashes generated during 5 seconds:');
console.log(count)
```

E: Ed25519-C/C++ Node.js-Addon-test

```
'use strict';
var test = require('tape')
var ed = require('./')
var bindings = require('./build/Release/supercop.node')
var startTime = new Date().getTime();
var count = 0;
var TD = 0;
test('generate, sign, and verify', function (t) {
  var endTime = new Date().getTime();
  TD = endTime  - startTime;
  while ((endTime  - startTime) < 5000) {
  t.plan(7)
  var seed = ed.createSeed()
  t.equal(seed.length, 32)
  var kp = ed.createKeyPair(seed)
  t.equal(kp.publicKey.length, 32)
  t.equal(kp.secretKey.length, 64)
  var msg = 'whatever'
  var sig = ed.sign(msg, kp.publicKey, kp.secretKey)
  var xsig = xmod(sig)
  var xmsg = xmod(msg)
  var xpk = xmod(kp.publicKey)
  t.ok(ed.verify(sig, msg, kp.publicKey))
  t.notOk(ed.verify(xsig, msg, kp.publicKey))
  t.notOk(ed.verify(sig, xmsg, kp.publicKey))
  t.notOk(ed.verify(sig, msg, xpk))
  var endTime = new Date().getTime();
  count++;}
console.log('Number of hashes generated during 5 seconds:');
console.log(count)
})

function xmod (buf) {
  var cp = Buffer(buf)
  cp[0] = ~cp[0]
  return cp}
```