Benchmarking Hash and Signature Algorithms

By Peng Wang, Carl Hua, and Michael Zochowski

Introduction

Hash and signature algorithms are the cornerstone of blockchain and crypto networks. Among other uses, they enable unforgeable identification and messaging, immutable recording of transactions, and efficient data storage. These properties make them a critical part of consensus, transaction verification, and, in proof-of-work networks, mining.

With the exception of proof-of-work (which, aside from extreme cases, is decoupled from the hash algorithm performance), the performance of the chosen hash and signature algorithms in a network has a major impact on performance. Every block and each transaction contained therein requires multiple signature verifications, hash verifications, and - for validators - a signature production. We estimate that the average crypto network node spends 25-65% of its CPU usage for hashing and signing, excluding mining. Optimizing these functions is one of the best ways to boost overall network performance, including transaction capacity and confirmation latency.

When we were architecting the Logos system, we were faced with the question of what signature and hashing algorithms best match our needs for high performance and minimal overhead. While there are many benchmarking results available on the web, they can vary significantly depending on the exact system environment. In particular, Logos uses proof-of-stake and a heavily optimized PBFT consensus scheme. We require multisignatures for efficient consensus, single signatures for transaction approval, and hashing for message digests, Merkle Trees, hashchains, and other data structures.

We therefore tested the performance of several hash algorithms and signature algorithms in our development environment last summer. For this benchmarking, we used AWS t2.2xlarge instances with 8 vCPUs from an E5-2686 v4 CPU at 2.30GHz running Ubuntu 16.04 with kernel 4.4.0.

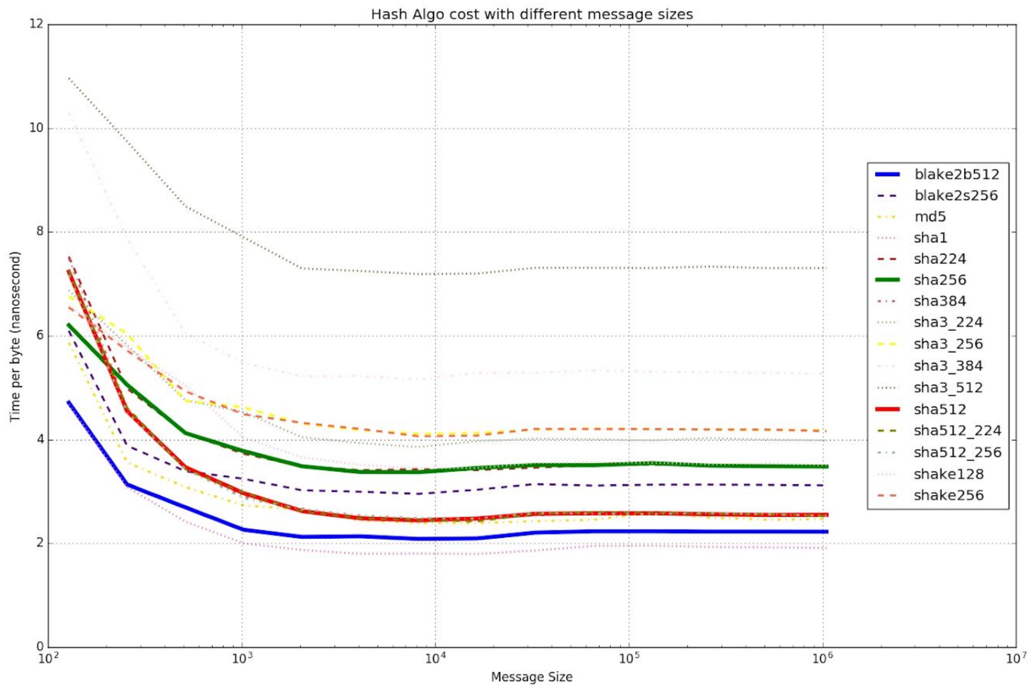
In this article, we summarize those benchmarking findings. You can find the raw code we used published on our Github here. This article is intended to be a technical overview and assumes familiarity with the various concepts and algorithms we tested.

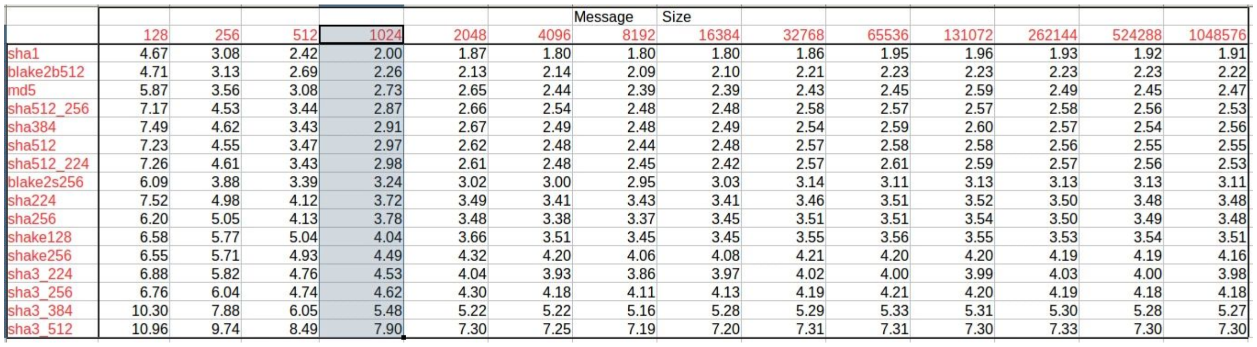
Hash Algorithms:

We tested some of the hash algorithms available in Openssl's libcrypto, version 1.1.1. The figure below is the per byte time (nanoseconds) of the hash algorithms with different message sizes. The x-axis is in log scale. The table underneath is the raw data, sorted by the column of message size 1024 bytes.

We can see that (1) for smaller messages up to 3KB, the cost per byte decreases when the message size increases, and for larger messages, the cost per byte is constant; and (2) the sha256 and sha512 have good performance, and blake2b512 is even faster.

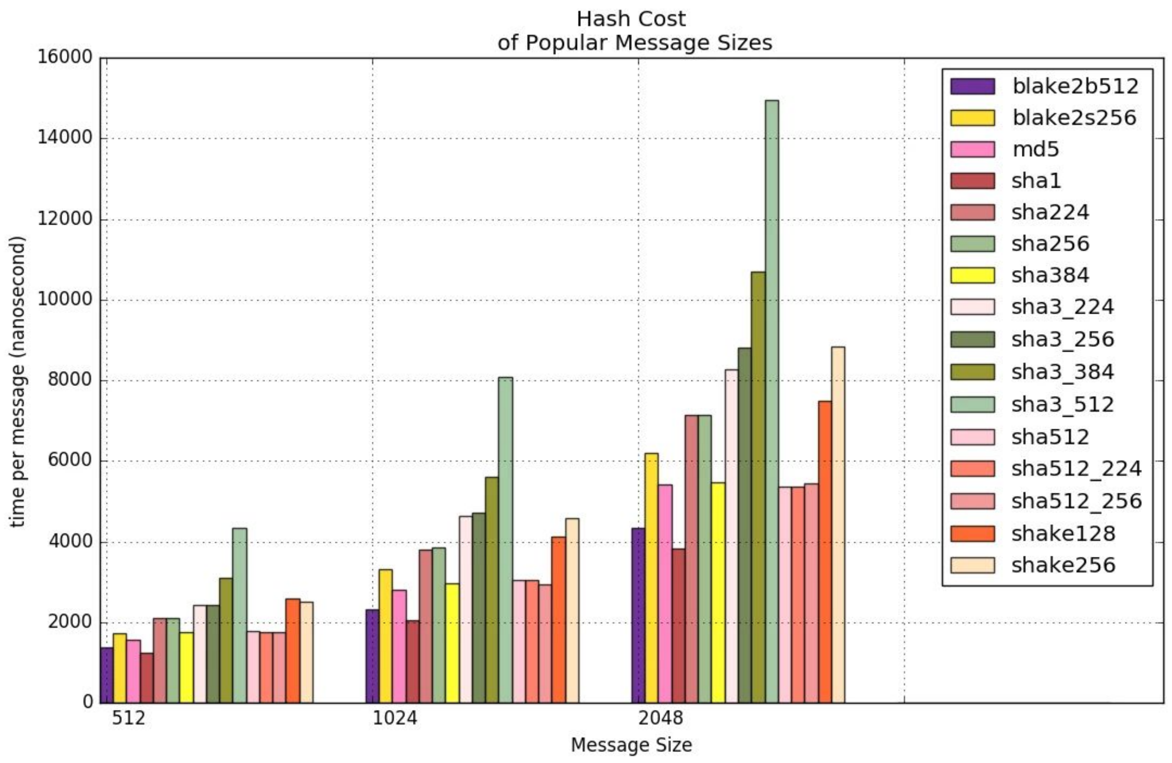
Please note that sha1 and md5 were included only for reference, and they were not considered for use in Logos due to their security issues.





							Message	Size						
	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576
sha1	4.67	3.08	2.42	2.00	1.87	1.80	1.80	1.80	1.86	1.95	1.96	1.93	1.92	1.91
blake2b512	4.71	3.13	2.69	2.26	2.13	2.14	2.09	2.10	2.21	2.23	2.23	2.23	2.23	2.22
md5	5.87	3.56	3.08	2.73	2.65	2.44	2.39	2.39	2.43	2.45	2.59	2.49	2.45	2.47
sha512_256	7.17	4.53	3.44	2.87	2.66	2.54	2.48	2.48	2.58	2.57	2.57	2.58	2.56	2.53
sha384	7.49	4.62	3.43	2.91	2.67	2.49	2.48	2.49	2.54	2.59	2.60	2.57	2.54	2.56
sha512	7.23	4.55	3.47	2.97	2.62	2.48	2.44	2.48	2.57	2.58	2.58	2.56	2.55	2.55
sha512_224	7.26	4.61	3.43	2.98	2.61	2.48	2.45	2.42	2.57	2.61	2.59	2.57	2.56	2.53
blake2s256	6.09	3.88	3.39	3.24	3.02	3.00	2.95	3.03	3.14	3.11	3.13	3.13	3.13	3.11
sha224	7.52	4.98	4.12	3.72	3.49	3.41	3.43	3.41	3.46	3.51	3.52	3.50	3.48	3.48
sha256	6.20	5.05	4.13	3.78	3.48	3.38	3.37	3.45	3.51	3.51	3.54	3.50	3.49	3.48
shake128	6.58	5.77	5.04	4.04	3.66	3.51	3.45	3.45	3.55	3.56	3.55	3.53	3.54	3.51
shake256	6.55	5.71	4.93	4.49	4.32	4.20	4.06	4.08	4.21	4.20	4.20	4.19	4.19	4.16
sha3_224	6.88	5.82	4.76	4.53	4.04	3.93	3.86	3.97	4.02	4.00	3.99	4.03	4.00	3.98
sha3_256	6.76	6.04	4.74	4.62	4.30	4.18	4.11	4.13	4.19	4.21	4.20	4.19	4.18	4.18
sha3_384	10.30	7.88	6.05	5.48	5.22	5.22	5.16	5.28	5.29	5.33	5.31	5.30	5.28	5.27
sha3_512	10.96	9.74	8.49	7.90	7.30	7.25	7.19	7.20	7.31	7.31	7.30	7.33	7.30	7.30

Messages in Logos are typically in the range of 512B to 2KB. The next figure shows the cost of hashing 512B, 1KB, and 2KB messages. Please note the y-axis is per message time. This clearly confirms the intuition that when the message size doubles, cost also roughly doubles.



We chose blake2 for Logos's hashing algorithm due to its speed. We specifically use blake2b256 because it targets 64 cpu architecture and provides 128-bit security.

In addition to its use in data structures and message digests, hashing algorithms are used in Logos for a minor proof-of-work spam prevention mechanism. For this role, we additionally considered Scrypt, Balloon, and Argon2id, but exclude them here for simplicity.

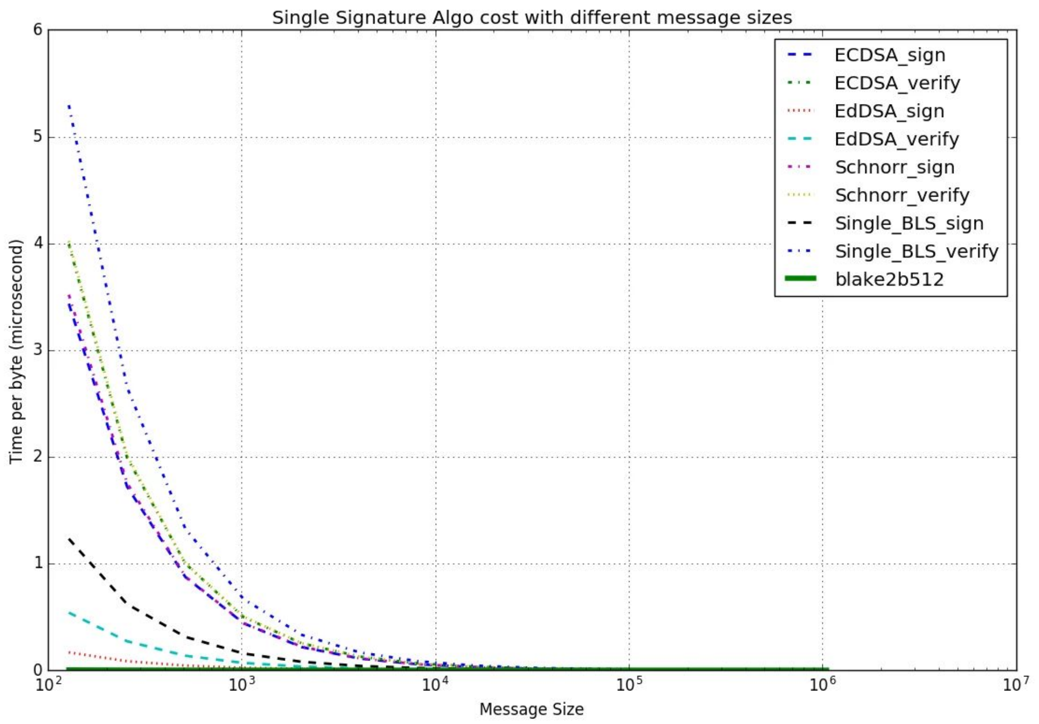
Signature Algorithms

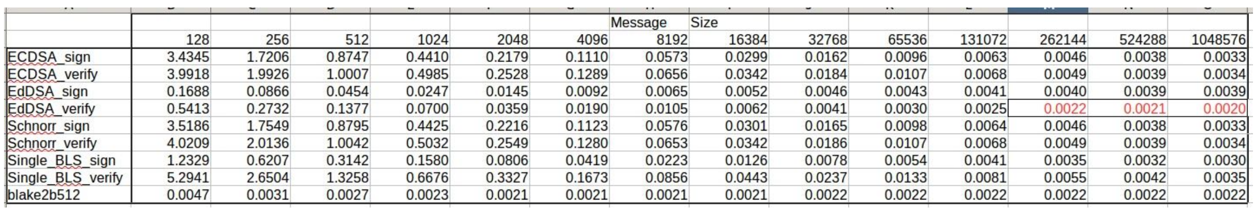
We tested the performance of the following signature algorithms: ECDSA (Openssl), EdDSA (Nano), EC-Schnorr (Zilliqa), and BLS (Keep).

Single signatures

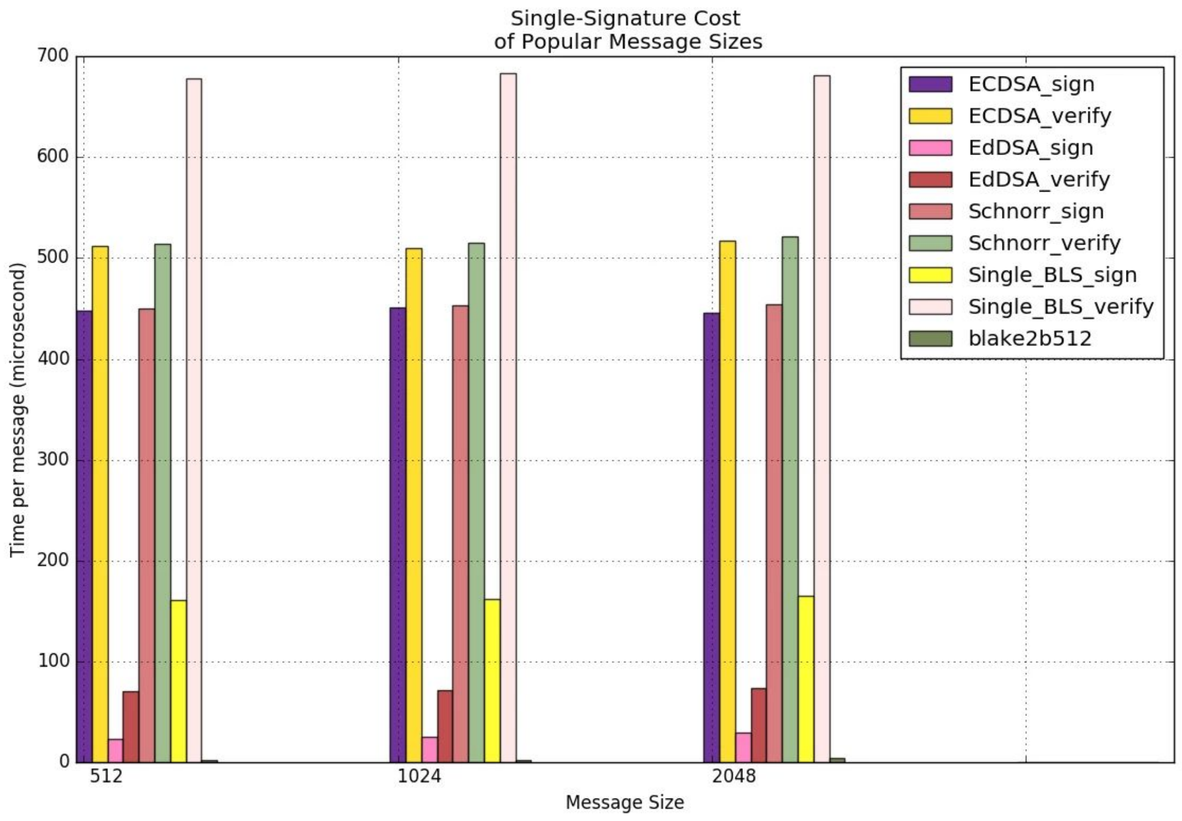
First, we show the cost of single signatures in the figures below. Similar to the previous section, we have the "Time per byte" figure for all message sizes, its raw data table, and the "Time per message" figure for 512B, 1KB, and 2KB messages, where time is measured in microseconds. We can see that (1) EdDSA is very fast, (2) for any algorithm, the cost of signing (verifying) 512B, 1KB, or 2KB messages are about the same.

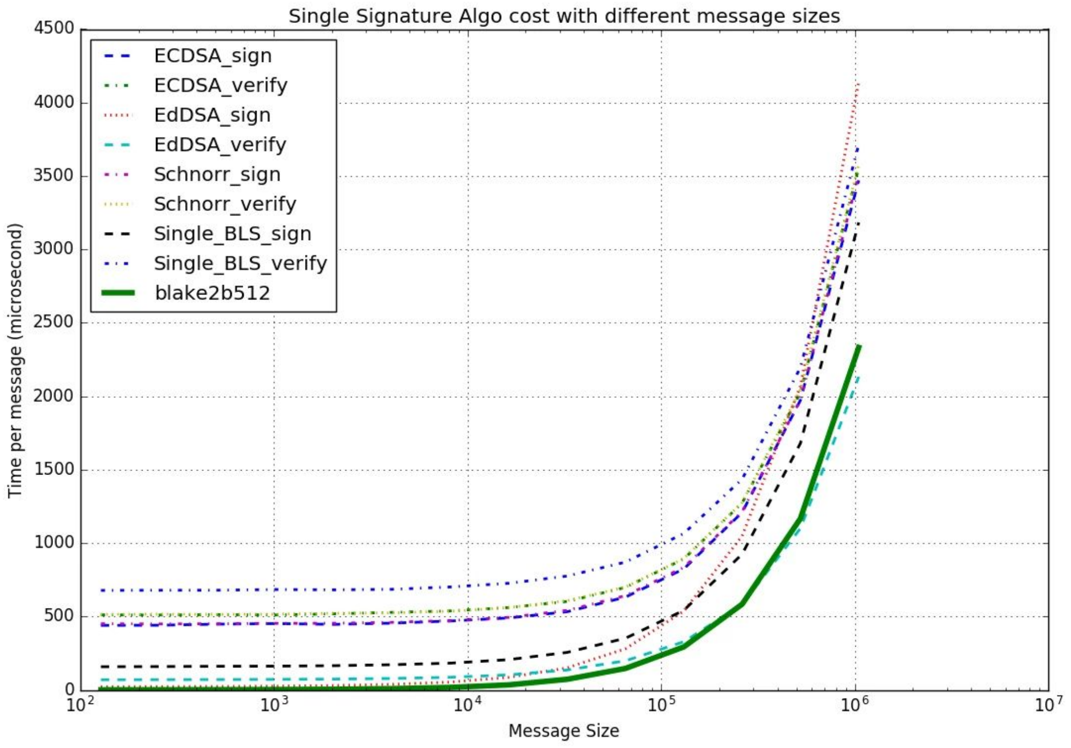
In general, a signature algorithm signs the digest of messages. I.e., a message is hashed to a fixed size digest, then the signature is computed on the digest. So the cost includes two parts (1) hashing cost which grows with message size, (2) signature operation cost which is constant. This is clearly shown in the last figure in this section. For this reason, we included blake2b512 in all the figures as a reference. In general, the cost of (2) significantly dominates that of (1).





							Message	Size						
	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576
ECDSA_sign	3.4345	1.7206	0.8747	0.4410	0.2179	0.1110	0.0573	0.0299	0.0162	0.0096	0.0063	0.0046	0.0038	0.0033
ECDSA_verify	3.9918	1.9926	1.0007	0.4985	0.2528	0.1289	0.0656	0.0342	0.0184	0.0107	0.0068	0.0049	0.0039	0.0034
EdDSA_sign	0.1688	0.0866	0.0454	0.0247	0.0145	0.0092	0.0065	0.0052	0.0046	0.0043	0.0041	0.0040	0.0039	0.0039
EdDSA_verify	0.5413	0.2732	0.1377	0.0700	0.0359	0.0190	0.0105	0.0062	0.0041	0.0030	0.0025	0.0022	0.0021	0.0020
Schnorr_sign	3.5186	1.7549	0.8795	0.4425	0.2216	0.1123	0.0576	0.0301	0.0165	0.0098	0.0064	0.0046	0.0038	0.0033
Schnorr_verify	4.0209	2.0136	1.0042	0.5032	0.2549	0.1280	0.0653	0.0342	0.0186	0.0107	0.0068	0.0049	0.0039	0.0034
Single_BLS_sign	1.2329	0.6207	0.3142	0.1580	0.0806	0.0419	0.0223	0.0126	0.0078	0.0054	0.0041	0.0035	0.0032	0.0030
Single_BLS_verify	5.2941	2.6504	1.3258	0.6676	0.3327	0.1673	0.0856	0.0443	0.0237	0.0133	0.0081	0.0055	0.0042	0.0035
blake2b512	0.0047	0.0031	0.0027	0.0023	0.0021	0.0021	0.0021	0.0021	0.0022	0.0022	0.0022	0.0022	0.0022	0.0022





We decided to use EdDSA for Logos since it was the most efficient signature scheme and has a number of robust, open source implementations.

Multisignatures

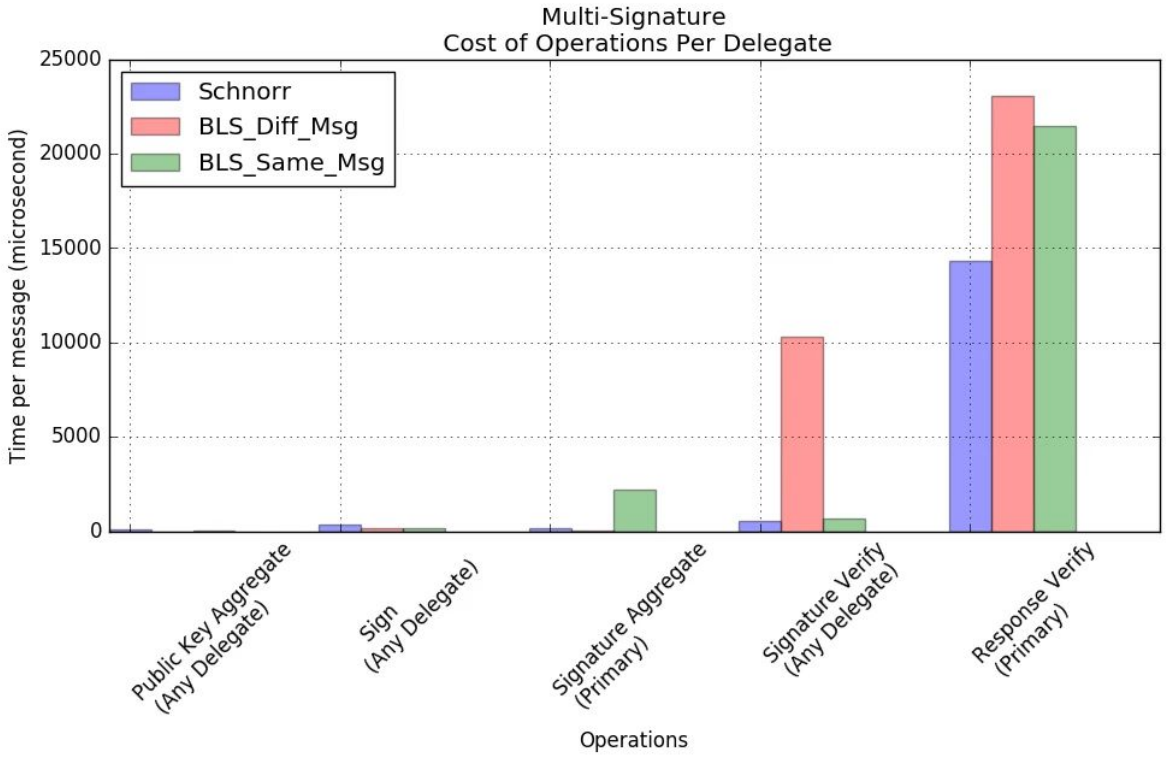
Transaction validation in Logos involves a set of 32 elected delegates running consensus instances in parallel. Each consensus instance involves a batch of transactions and is run by a primary delegate with 31 backups. A supermajority (greater than ⅔) of delegates must sign off on each batch of transactions for them to be confirmed. Safety is strongly guaranteed through a combination of mathematically proven distributed algorithms and game theory.

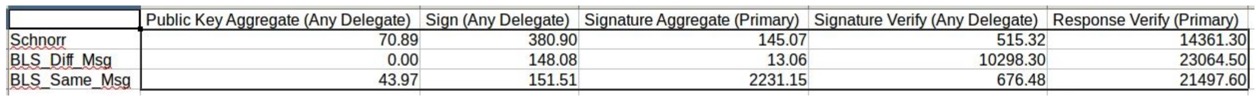
If Logos were to implement this system using single signatures, validating that a batch (and thus the contained transactions) is valid as well as storing the network history would be extremely inefficient in both CPU and storage space since each batch involves at least 22 signatures. We therefore use a multisignature scheme to make signature size constant, regardless of the number of individual signatures it contains.

We tested three multisignature algorithms. The Schnorr multisignature is a two-round interactive algorithm. The BLS_Diff_Msg is a simple aggregation algorithm included in herumi's code "github.com/herumi". It does not have a public key aggregation algorithm and is not secure when the messages could be the same. The BLS_Same_Msg is built on top of herumi's code. It has a public key aggregation algorithm, and it requires all the messages are the same.

In this section, all the messages are 512B, since we already know message sizes only affect the hashing cost of signatures, from the previous section. The figure below shows the time cost of the operations. Some observations:

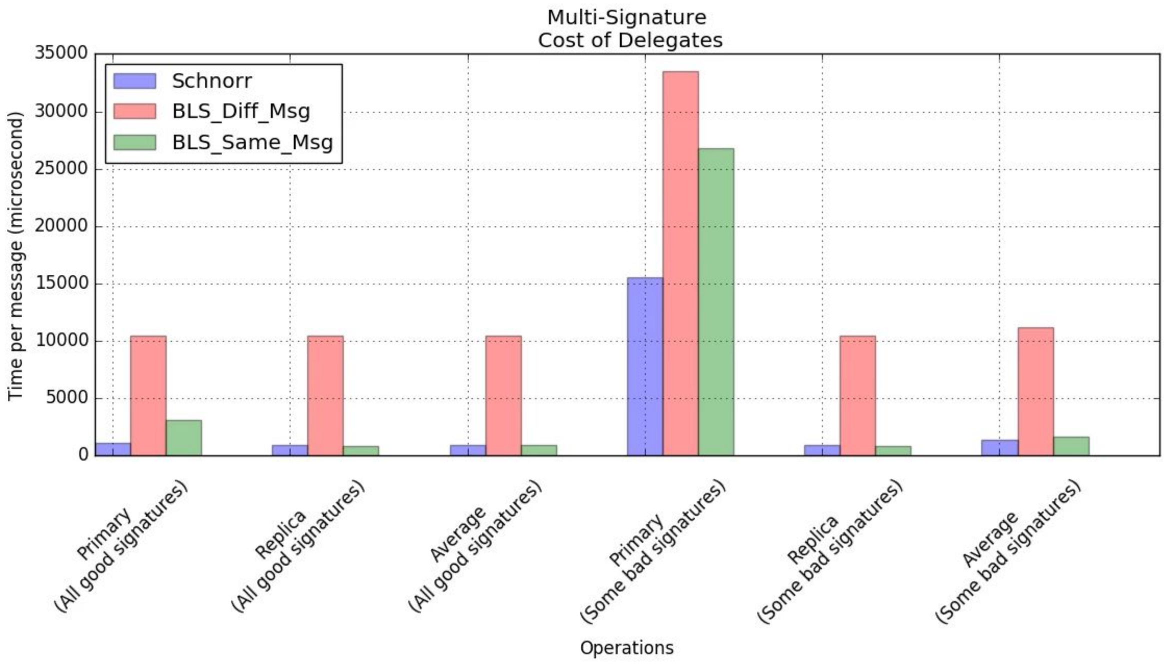
1. "Public key aggregation" cost is small, and is a one time cost for the same set of delegates.
2. The Schnorr algorithm's signing operation is a bit higher than BLS.
3. BLS_Diff_Msg's signature aggregation cost is very low and BLS_Same_Msg's signature aggregation cost is much higher.
4. The signature verification cost of BLS_Diff_Msg is very high.
5. No matter which algorithm, if some of the input to the signature aggregation function are corrupted, the cost of finding the corrupted ones (marked as Response Verify in the figure) is very high.

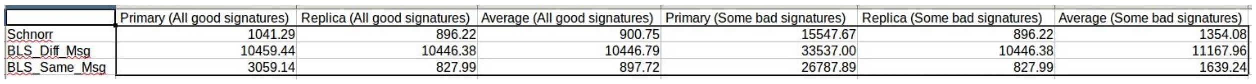




	Public Key Aggregate (Any Delegate)	Sign (Any Delegate)	Signature Aggregate (Primary)	Signature Verify (Any Delegate)	Response Verify (Primary)
Schnorr	70.89	380.90	145.07	515.32	14361.30
BLS_Diff_Msg	0.00	148.08	13.06	10298.30	23064.50
BLS_Same_Msg	43.97	151.51	2231.15	676.48	21497.60

The next figure is the cost of the delegates of one multi-signature generation. We considered two cases, (A) all 31 delegates returned good individual signatures (or equivalent in Schnorr) to the primary, (B) some (<1/3) individual signatures are corrupted. Other cases such as missing signatures, or too many corrupted signatures which resulted in a failed round should have less cost as compared to cases (A) and (B).





	Primary (All good signatures)	Replica (All good signatures)	Average (All good signatures)	Primary (Some bad signatures)	Replica (Some bad signatures)	Average (Some bad signatures)
Schnorr	1041.29	896.22	900.75	15547.67	896.22	1354.08
BLS_Diff_Msg	10459.44	10446.38	10446.79	33537.00	10446.38	11167.96
BLS_Same_Msg	3059.14	827.99	897.72	26787.89	827.99	1639.24

Thus, Schnorr outperforms BLS in CPU space, but BLS is within an order of magnitude. What these results do not show is the cost of messaging. In particular, a Schnorr multisignature requires one more round of messages between the aggregator and single signers than BLS, which complicates the core consensus protocol and imposes a significant time overhead due to latency and communication. Assuming a 200ms latency (a multisignature is limited by the highest latency signer), the Schnorr multisignature would have an additional latency of 200,000 microseconds. This means that BLS is drastically more time efficient than Schnorr in a real world setting. In addition, BLS requires only half the data as Schnorr for the equivalent security level. For example, the signature size of Schnorr multisignature scheme is 64 bytes for 128-bit security, while the signature size of BLS is only 32 bytes.

For these reasons, among others, we decided to use BLS multisignatures in Logos. To enable signature aggregation, we extended publicly available implementations based on Boneh et al. We'll release that implementation publicly when we open-source the Logos core codebase.

More Benchmarking Data: eBACS

ECRYPT Benchmarking of Cryptographic Systems (eBACS) is a great source of bench-marking results of many cryptographic algorithms.