STAR and verifiability of shares

Sofía Celi

1 Notation

We recall first the main participants, parameters, cryptographic tools, and notation that we use when describing both the STAR and POPLAR protocols.

- k is the threshold used for performing server-side aggregation.
- n is the total report size submitted by C clients.
- C is the set of all clients $\{C_i\}_i \in [n]$.
- S is the aggregation server.
- O is the randomness server used in STAR.
- m is a message to secret-share.
- t is an integer $\in N$ that states in POPLAR that a string σ appears in a list $(a_1, \dots a_c)$ more than t times.
- σ is a string to search for in a list.
- l is the length of σ.

2 Overview of STAR [DSQ+22]

Each client constructs a ciphertext by encrypting their measurement (and any auxiliary data) using an encryption key derived deterministically from i) any randomness present in the client measurement and ii) additional randomness provided by a "randomness server". This randomness server never learns client values or inputs. The client then sends: i) the ciphertext; ii) a k-out-of-n secret share of the randomness used to derive the encryption key; and iii) a deterministic tag informing the server which shares to combine. The aggregation server groups reports with the same tag, and recovers the encryption keys from those subsets of size $\geq K$. Thus, the server learns all the measurements that are shared by at least K clients (along with any auxiliary data).

2.1 Secret Sharing scheme in STAR

In STAR, we use a secret sharing scheme called "adept secret-sharing (ADSS)" [BDR20]. This scheme provides the following properties (we follow the notation from [BDR20]):

- *Reproduciability:* the ability to recompute a share, or a vector of shares, as long as you still have the secret. The scheme is deterministic.
- *Authenticity:* The recovery procedure either fails or recovers the secret originally associated to it. This means that shares are *committed* to one specific secret (and cannot be committed to anything else). The threshold structure on each share cannot be modified as well.
- *Error-correction:* The recovery procedure "heals itself" if invalid shares are present (if either they are corrupted or new shares are added). Notice that this property is slow and can be removed from the scheme.

The scheme provides two structures to achieve sets of these properties:

- A secret sharing scheme with an **AX** transformation: This transforms any secret-sharing scheme (as Shamir's Secret-Sharing one [Sha79], for example) into a ADSS with reproduciability and authenticity.
- A secret sharing scheme with an **AX** and **EX** transformations: This transforms any secret-sharing scheme (as Shamir's Secret-Sharing one, for example) into a ADSS with reproduciability, authenticity and error-correction.

The first structure (ADSS with **AX**) is highly efficient: sharing an m-byte message M will take $O(m)$ time and, more concretely, about the amount of time to symmetrically encrypt and hash M. This assumes a fixed number of shareholders C, a fixed access-structure encoding A, a fixed tag T, and fixed scheme parameters. Concretely, to share M, one will need to apply a hash function like SHA-256 to a string that is $|A| + |T| + k$ bits longer than M (likely $k \in \{128, 256\}$), run a blockcipher like AES in counter mode to generate a pad that is k-bits longer than $|M|$, and run a sharing under S_1/S_2. Message recovery takes the same time as sharing.

The second structure (ADSS with **AX** and **EX**) does not impact sharing, but does impact recovery. It is exponential on the amount of shares passed to the recover procedure (n shares that will be divided in k subsets). In the worst case, it inspects 2^n when there are corrupted shares (but not when there are omitted shares).

In the STAR protocol (in the original ACM-CCS publication), we chose to use ADSS with **AX**n.

3 Malicious shares and malicious clients

Not having *error-correction* (or verifiability) opens an attack against the STAR protocol. The assumption of this attack is that there exists a set x of malicious clients that corrupt j amount of shares. In turn, this means that the recovery procedure will be unable to pinpoint which share is invalid (corrupted): the recovery procedure will halt and the whole batch of k size will be discarded. This gives the possibility for malicious clients to perform DoS attacks with the goal of discarding sets of honest measurements. Notice that even if the attacker corrupts an small set of shares, they can still cause tangible disruption to the whole set of measurements. Note that in this attack we assume that clients are corrupted but the dealer (the browser, for example) is not.

There are ways to solve this:

- Use ADSS with **AX** and **EX**.
- Use a secret sharing scheme with verifiability (Feldman's scheme [Fel87] or Pedersen scheme [Ped92]).
- Perform **EX** with a different construction which is the subject of a publication under review. In this work we arrive to a construction that achieves $O(\log n)$.

3.1 ADSS with AX and EX

Informal description. In [BDR20], correctness expects that *Recover*(*shares*) (where $|shares| = k$) returns $(m, sub_)$, where *sub* is an authorized subset of some sharing of m, and m is the recovered message. However, if *sub* is not an authorized subset of some sharing of m (there are corrupted shares in *sub*, there are two possibilities to take: *error-detect* or *error-correct*.

Error-detection will allow for the scheme to return $\perp$, which signals that something went wrong, and m cannot be recovered. However, Error-detection comes with a liability: it enables the adversary to thwart message recovery by corrupting a single share (as stated, this also allows for mounting a DoS attack).

Error correction (*Errx*), on the contrary, seeks to recover from errors whenever the algorithm can. It not only detects an error but also "heals" from it.

What [BDR20] proposes is to create a scheme that checks that each share presented to *Recover*(S) name the same access-structure encoding A, the shares all have the same tag T; and the shares include all those in K if K is a set of shares. The worst case is exponential (it inspects 2^n) in the case that corrupted shares are placed at the end of each k subset. Notice that the error correction mechanism "corrects" the shares as it iterates over subsets of k size until it finds an honest subset and discards any share that does not form a honest subset of k size.

3.2 ADSS with AX and verifiability

Another proposal is to "detect" corrupted shares by providing with a "proof" of each share's honesty. There are some mechanisms for doing so (like providing a signature with each share) [CGMA85, BGW88, CCD88, RB89].

We note here the ones based on polynomials due to their nice algebraic properties, which allow us to define *hiding* properties (the fact that the values of a polynomial of degree d in less than d points give no information of its value in another point), *error correction* properties (the fact that there exists a unique polynomial of degree d given $d + 2d_a + 1$ values where at most d_a of them are incorrect), and they are *homomorphic*.

We present here both Feldman's scheme and Pedersen' scheme. Note that in both schemes the dealer can commit either to the coefficients of the polynomial $f(x)$ or to the client shares $f(a_i)$. Pedersen's commitments will be of the form $g^s h^\rho$, since g^s (used in Feldman's scheme) is not considered a real commitment as the hiding property is not satisfied. Note, though, that Feldman's scheme can be implemented with Pedersen's commitments.

- *Feldman's scheme [Fel87]*: the scheme utilizes homomorphic relationships between values and their encryptions, and using cyclic groups. The communication and computation complexity are small, $O(n\lambda)$ and $O((n \log n + \lambda)(n * \lambda \log \lambda))$ respectively, where λ is the security parameter. In the case of using discrete-log in finite-fields, the computational complexity is impacted by exponentiation: g^s can be performed in at most $2|x| \leq 2|p| = 2\lambda$ multiplications mod p and at most λ additions. The speed for the multiplication can be considered as $O(\lambda \log \lambda)$. The more costly part of this algorithm is checking for valid shares.

- *Pedersen's scheme [Ped92]*: this scheme is based on the previous one but no leakage is present. Notice that we have to compute k commitments in order to verify a single share. This requires less than $2 * |q| * k$ multiplications modulo p or approximately $2k$ multiplications per bit of the message, if every element in Z_q can be chosen as the message. The verification requires $k - 1$ exponentiations modulo p and the computation of one commitment. This can be done in less than: $2*|q|*(k-1)+2*|q|+(k-1) \approx (2*|q|*1)*k$ multiplications. This is considered, however, a pessimistic estimate as many of the exponents in the exponentiations are rather small.

4 Time complexity of a "verifiable" STAR

STAR can be divided into these operations:

1. Randomness extraction: which can be either OPRFs operations, hash operations or any other deterministic randomness extraction (AES-based, for example). The time complexity of this phase depends on the scheme chosen.

2. Sorting: when executing the recovery procedure, the STAR scheme can:

   - Wait for receiving n shares and sort them into subsets of k size according to the *tag* value received in each share.
   - Place each share in a subset of k size (in which all other shares share the same *tag*) the moment it is received.

   In order to properly sort the received shares, an efficient sorting algorithm can be used, such as **Merge sort** (which has a best, average and worst time complexity of $O(n * \log(n))$).

3. Verifiable Secret sharing scheme: a verifiable secret sharing scheme comprises of:

   - Share creation: The operation corresponds to the "splitting" of a input secret into n shares, where k shares are enough to recover the secret message. This operation is very efficient.

- Recovery: The operation corresponds on receiving a subset of k size of shares, and recovering the underlying secret message.
- Verifiability: this procedure can be applied to a single share, shares inside a subset of k size, or all shares on the set of n size. It verifies that said share has not been corrupted.

4.1 Time complexity of Verifiable Secret Sharing

The algorithm consists of two phases:

- **Commitments creation:** Generate the needed commitments that "prove" that a share is "honest". This procedure is generated (it runs) once for a set of n shares. It is linear on the size of k (it generates k commitments for a set of n shares).
- **Verification of share:** Verify a single share by attesting that the commitments are "honest". It is linear on the size of k. Notice that this phase can be expanded to verify a whole subset/set of shares, in which case, it is $O(n * k)$ (for verifying the set of n shares) or $O(k^2)$ (for verifying a subset of k size).

As stated, this procedure can be applied to a single share, shares inside a subset of k size, or all shares on the set of n size. In the case of verifying shares inside a set or subset, we can define:

- **Worst-case complexity:** This case occurs when a corrupted share is placed at the end of the set/subset. The cost is: cost of verifying a single-share ($O(k)$) * (size of set $\vee$ size of subset): $O(k * (n \vee k))$
- **Average-case complexity:** If each share is equally likely to be corrupted, then the scheme has an average case of n in simple terms. Note that the average case can be affected if the corruption probabilities for each share vary (which is the case here as only a subset of clients can be considered malicious): in our case, then, the average case depends on the probability of the attacker of corrupting a set of shares and of the network on delivering them in a specific order.
- **Best-case complexity:** This case occurs when there is only one corrupted share per set/subset and it is placed at the start of the set/subset. The cost is: cost of verifying a single-share ($O(k)$) * 1 * number of sets/subsets: $O(1 * (1 \vee |\{x \subset n : |x| = k\}|))$

The verification algorithm depends on the size of k, the threshold, as it generates k commitments that are verified on each share.

5 "Smart" algorithm for STAR-VSS

ADSS with **AX** and verifiability (in STAR-VSS) can be executed at any time:

- Per each single share.
- Per each subset x of size k.
- Against the whole set y of size n.

We can therefore create a "smart" algorithm that:

1. Performs the recovery functionality first on a subset x_i of k size. If it fails on the subset, it runs the verification of that single subset x_i.
2. The verification algorithm will remove the invalid shares (the subset r of x_i), and return a subset of size $k - |r|$.
3. The returned subset of size $k - |r|$ can be used to construct a k size subset (by fetching w shares from the set y of n size so that $k - |r| + |w| = k$), and perform recovery again.

Clients	Computation	Bandwidth
100k	13.8 min	6.5 GB
200k	27.2 min	13.1 GB
400k	53.8 min	26.2 GB

Table 1: Total cost for servers side aggregation: searching for top-900 heavy hitters, 256-bit length (l) of σ.

6 Lightweight Techniques for Private Heavy Hitters: POPLAR [BBC+21]

A different proposal for aggregating measurements is POPLAR [BBC+21], which uses incremental distributed point functions, a cryptographic primitive that builds on standard distributed point functions (DPFs). In it, each client holds a l-bit string and a set of servers aggregates them. If each client holds an l-bit string, with plain DPFs, the client computation and communication costs would grow as l^2. With incremental DPFs, this cost falls to linear in l. For the applications they note, they set $l \approx 256$, so this performance improvement is substantial.

As stated in [BBC+21], POPLAR increases its computational and communication costs when l-bit string becomes longer, as each client sends each server an all-prefix DPF key with domain size l. Each key is approximately $\lambda * l \log C$ bits in length, where C is the number of clients and $\lambda \approx 128$ is the size of a PRG seed.

In some applications, the system might want to learn the most popular values over long strings. For example, an operating-system vendor might want to learn the set of popular software binaries running on clients' machines that touch certain sensitive system files. In this application, client i's string $a_i \in 0, 1^n$ is an x86 program, which could be megabytes long. So for this application, $l \approx 2^{20}$.

When l is much bigger than λ, the authors suggest to use hashing to reduce the client-to-server communication from $\approx \lambda * l \log C$ bits down to $\approx \lambda^2 \log C + l$ bits and the round complexity from $\approx l$ to $\approx \lambda$.

7 Comparison

Comparing both the STAR-VSS protocol and the POPLAR protocol is a difficult tasks as both protocols grow depending on different parameters: POPLAR increases in regards to l, the size –in bits– of the string to search for, while STAR-VSS increases in regards to k, the threshold parameter. In POPLAR, communication and computational costs are determined by l, and can be considered in parallel as servers need to communicate with each other in order to aggregate (this assumption, however, should not be made from the client side). In STAR-VSS, the size of the string to secret-share is not taken into account as, when using Elliptic-Curve Cryptography (ECC), it will be hashed into a scalar representation. There is also no additional communicational cost in STAR from the server side, as the protocol is a single-server one.

The authors of POPLAR report in their IEEE S&P talk [CG21] the numbers shown on Table 1. The computation times shown are parallelizable.

We performed some benchmarks in Rust of the secret sharing scheme (Shamir Secret Sharing) with verifiability (the Feldman's scheme): the code is not optimized. The Rust implementation can be found here: `https://github.com/claucece/secret-sharing-extra`. We defined the following parameters: threshold (which is k, the subset size), and report size (which is n, the total size of the measurements reported). In all cases by *secret* we used a string of 32 bytes in size. We report numbers when using curve25519/ristretto (shown on Table 2) and Sec256k1 (shown on Table 3) for the field and elliptic curve operations. We ran our benchmarking on a MacBook Pro with arm64, Darwin Kernel Version 22.3.0, Apple M1 Max chip. We are using Rust with version 1.68.0.

As seen on the tables, the numbers increase in regards to k. This is shown in Figure 1 for $n = 256$ over curve25519/Ristretto, and on Figure 2 (which shows that with different n, the k-growth is proportional: the Figure is up to $n = 1024$ for readability). However, the growth is also depend on the elliptic curve chosen: sec256k1 is faster than curve25519 as seen in Figure 3 for $n = 256$, and in Figure

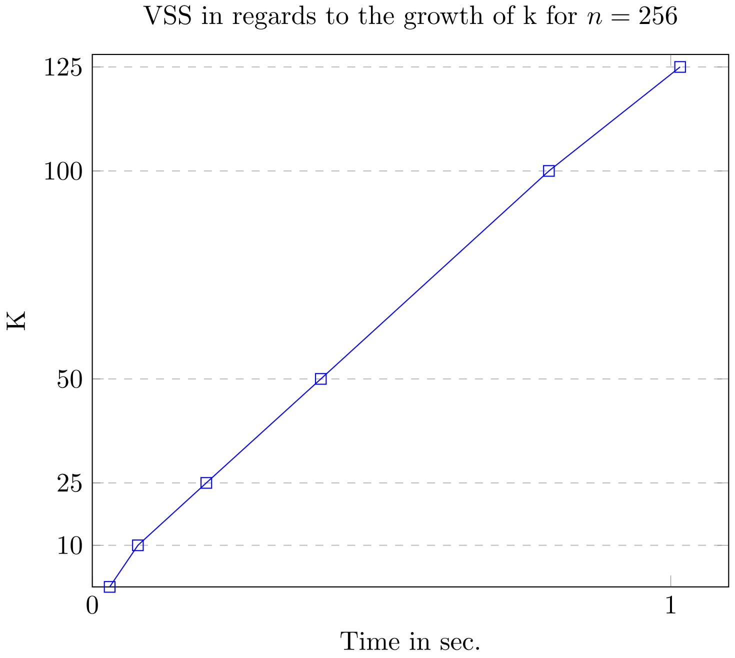

Figure 1: VSS scheme in regards to the growth of k for $n = 256$ when using Curve25519/Ristretto.

4 for $n = 2048$. Hence, the VSS scheme is efficient when k is small. In the case of STAR, we can use this assumption as k does not need to be long (it can be $k \geq 10$ and ≤ 50), and we can immediately process subsets of k size as soon as they arrive to the single-server.

For POPLAR [BBC+21], we are using the measurements framework as defined in `https://github.com/henrycg/heavyhitters`. The code compiles only for a older Rust version (we are using, hence, 1.47.0) on a older MacBook Pro 12.3.1 with 2.3 GHz Dual-Core Intel Core i5 (model I5-7360U) with x86_64: we are using an older OS as older versions of Rust don't compile on the newer versions of the OS. This makes the POPLAR measurements perhaps not as accurate as the ones taken for the VSS scheme.

Note that the times reported in Table 4 correspond to the end-to-end performance test of Poplar. It shows the running time from the moment after the servers collect the last incremental DPF keys from the clients until the servers produce their output. It tests both over 256-bit length strings and 512-bit length strings (we couldn't compile the code with longer strings). As shown in the table, the costs increase as l increases. POPLAR performs well for small strings (small l) and with the usage of parallelization. The performance can be better understood in Figure 5.

8 Future work

While we have presented some performance measurements, the POPLAR measurements need to be better specified: they should work with the same Rust version as the VSS measurements and on the same OS. This is left as future ongoing work.

We have treated the VSS measurements as well on their own: we should measure them with the whole STAR-ADSS-VSS scheme. This needs first to adapt the formalization of [BDR20] to take into account this notion of verifiability, and measure only with the formalised algorithm. This is left as future ongoing work.

Currently, we are developing a verifiable scheme that arrives to worst/average case time of $O(\log n)$. It is left as future ongoing work to properly formalise it and measure it.

Report size (n)	**Threshold (k)**	**Verification Time**
256	10	0.078823
	25	0.19705
	50	0.39504
	100	0.78929
	128	1.0161
512	10	0.15879
	25	0.39705
	50	0.79414
	100	1.5918
	128	2.0291
768	10	0.23787
	25	0.59317
	50	1.1824
	100	2.3841
	128	3.0356
1024	10	0.31638
	25	0.78658
	50	1.5812
	100	3.1731
	128	4.0413
1280	10	0.39412
	25	0.99017
	50	1.9723
	100	3.9375
	128	5.0440
1536	10	0.47586
	25	1.1898
	50	2.3707
	100	4.7534
	128	6.2343
1792	10	0.56573
	25	1.4113
	50	2.8184
	100	5.6685
	128	7.2657
2048	10	0.63156
	25	1.5732
	50	3.1592
	100	6.3074
	128	8.1037

Table 2: Benchmarks for the secret sharing scheme with verifiability: it shows the time for verification of n shares using Feldman's scheme using Curve25519/Ristretto. Numbers are reported in seconds.

Report size (n)	**Threshold (k)**	**Verification Time**
256	10	0.039585
	25	0.084872
	50	0.16170
	100	0.31496
	128	0.39843
512	10	0.078473
	25	0.17081
	50	0.32648
	100	0.63951
	128	0.81151
768	10	0.11953
	25	0.25948
	50	0.49687
	100	0.96418
	128	1.2394
1024	10	0.15889
	25	0.34727
	50	0.65934
	100	1.2817
	128	1.6277
1280	10	0.19776
	25	0.43309
	50	0.83642
	100	1.6158
	128	2.0504
1536	10	0.23925
	25	0.52875
	50	0.99389
	100	1.9495
	128	2.4761
1792	10	0.28050
	25	0.62505
	50	1.1704
	100	2.2827
	128	2.9096
2048	10	0.31876
	25	0.69710
	50	1.3404
	100	2.6018
	128	3.3170

Table 3: Benchmarks for the secret sharing scheme with verifiability: it shows the time for verification of n shares using Feldman's scheme using curve sec256k1. Numbers are reported in seconds.

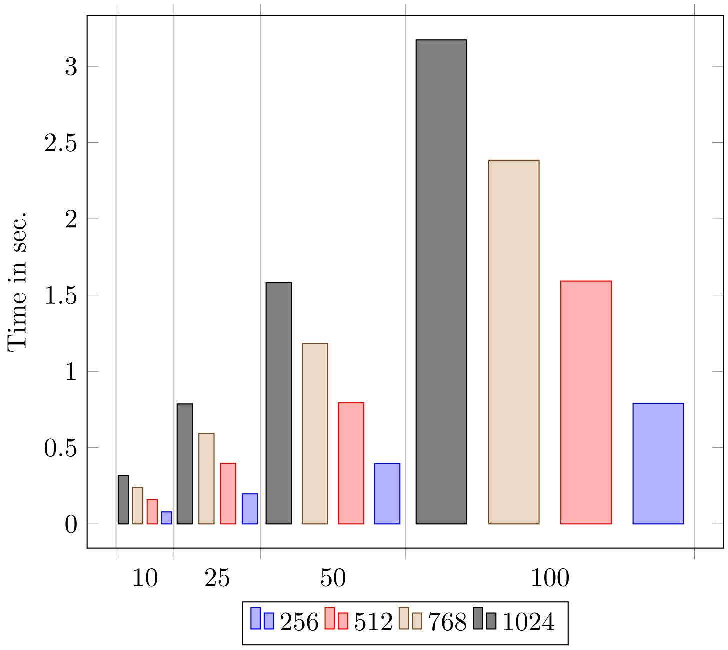

Figure 2: Comparison of benchmarks: x-axis states the threshold sizes, y-axis states the times. The numbers that relate to the colours represent the set size n. This is using the times reported when using Curve25519/Ristretto.

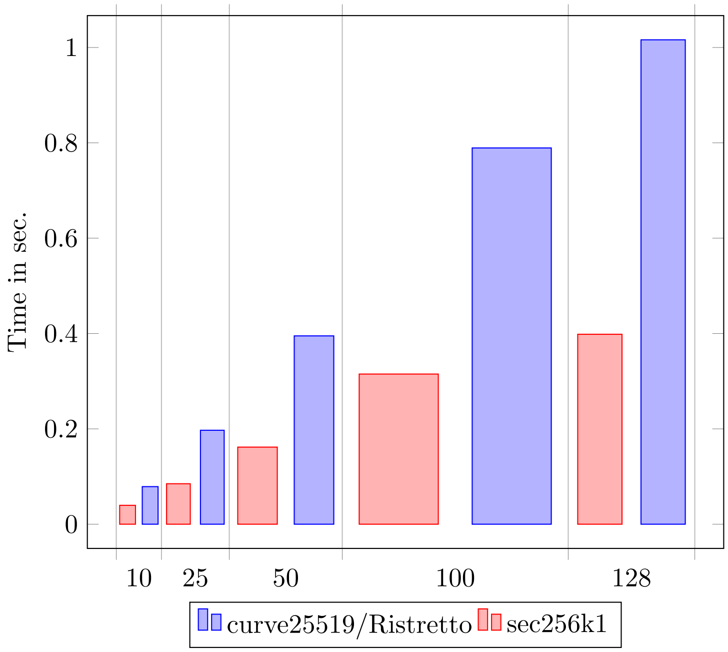

Figure 3: Comparison of benchmarks: x-axis states the threshold sizes, y-axis states the times in seconds. The numbers that relate to the colours represent the curve choice: Curve25519/Ristretto or sec256k1. The numbers are for $n = 256$.

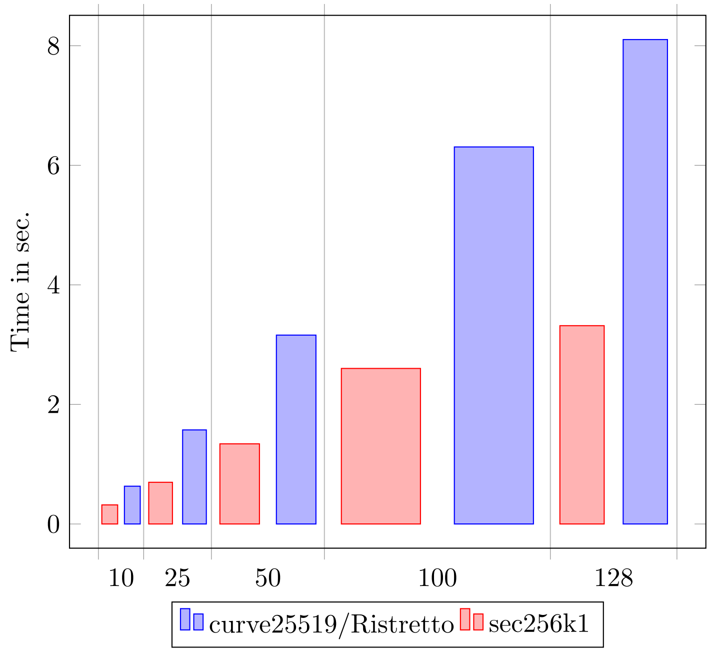

Figure 4: Comparison of benchmarks: x-axis states the threshold sizes, y-axis states the times in seconds. The numbers that relate to the colours represent the curve choice: Curve25519/Ristretto or sec256k1. The numbers are for $n = 2048$.

Input size	**Client requests**	**Threshold**	**Total Time**
256	**256**	0.1%	5.507831327
	512	0.1%	16.959929453
	768	0.1%	36.15255669
	1024	0.1%	55.856106997
	1280	0.1%	86.420286407
	1536	0.1%	126.5820803
	1792	0.1%	155.289137682
512	**256**	0.1%	13.881579577
	512	0.1%	38.576247141
	768	0.1%	76.272295041
	1024	0.1%	130.002528967
	1280	0.1%	170.99829304
	1536	0.1%	294.671993384
	1792	0.1%	332.42823375

Table 4: Bechmarks for the POPLAR scheme: the threshold here represents that more than 0.01% clients hold a specific string. The times are reported in seconds.

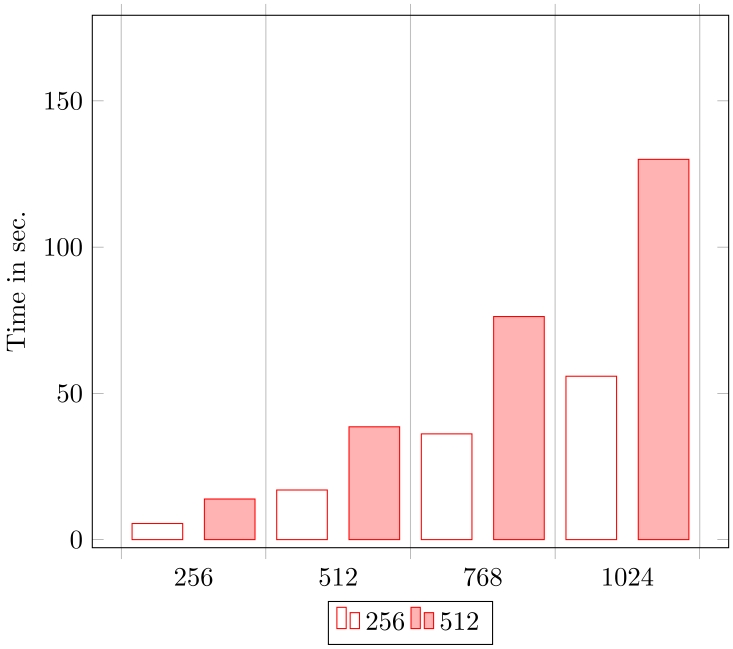

Figure 5: Comparison of benchmarks: x-axis states the number of client request, y-axis states the times (in seconds). The numbers that relate to the colours represent the input size l.

References

[BBC+21] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. pages 762–776, 2021.

[BDR20] Mihir Bellare, Wei Dai, and Phillip Rogaway. Reimagining secret sharing: Creating a safer and more versatile primitive by adding authenticity, correcting errors, and reducing randomness requirements. 2020(4):461–490, October 2020.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). pages 1–10, 1988.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). pages 11–19, 1988.

[CG21] Henry Corrigan-Gibbs. Lightweight techniques for private heavy hitters. https://www.youtube.com/watch?v=d2gvAPPFAPE, May 2021.

[CGMA85] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults (extended abstract). pages 383–395, 1985.

[DSQ+22] Alex Davidson, Peter Snyder, E. B. Quirk, Joseph Genereux, Benjamin Livshits, and Hamed Haddadi. STAR: Secret sharing for private threshold aggregation reporting. pages 697–710, 2022.

[Fel87] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. pages 427–437, 1987.

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. pages 129–140, 1992.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). pages 73–85, 1989.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.