Supplementary Information

Supplementary Material, Tables, and Figures

Security of homomorphic encryption and RLWE. Most widely-used homomorphic encryption schemes [6,5,12,9], including the BGV scheme we employed, rely on the hardness of a mathematical problem called the Ring Learning with Errors (RLWE) problem, *i.e.,* the ring version of Learning with Errors (LWE) problem. The hardness of the LWE problem has served as a common assumption due to its quantum reduction to a hard lattice problem proposed by Regev [21] and its robustness against various attacks [4,3]. Later, Lyubashevsky *et al.* [19] introduced the RLWE problem and demonstrated its reduction to the worst-case lattice problems over ideal lattices. Compared to LWE-based cryptographic schemes, RLWE-based schemes are more efficient, especially for power-of-2 cyclotomic rings [22].

To delve deeper into the RLWE problem, we introduce the following notation: the $\mathfrak{m}$-th cyclotomic polynomial $\Phi_{\mathfrak{m}}(X)$ of degree $\mathfrak{n}$, an integer-coefficient polynomial ring $\mathcal{R} = \mathbb{Z}[X]/\Phi_{\mathfrak{m}}(X)$, a modulus Q and the quotient ring $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R}$, an error distribution χ, and $\leftarrow$ for a uniform sampling.

Given a secret $s \in \mathcal{R}_Q$, the hardness of RLWE guarantees that the following two distributions in $\mathcal{R}_Q \times \mathcal{R}_Q$ are (computationally) indistinguishable under appropriately chosen parameters:

$$\mathcal{D}_s = \{(a, b) : a \leftarrow \mathcal{R}_Q, e \leftarrow \chi, b = as + e\}, \mathcal{D}_{\text{rand}} = \{(a, b) : a, b \leftarrow \mathcal{R}_Q\}.$$

As such, the secret s is masked using a pair of elements, which are indistinguishable from two uniform elements. To achieve the desired λ-bit security in RLWE-based HE schemes, parameters $(\mathfrak{m}, Q)$ need to be chosen such that the best-known LWE attacks require at least 2^λ operations. Concrete parameters are generally estimated following [2], which are also standardized in the white paper [1].

Experimental setup and HE parameters. We benchmarked SQUiD on an *n2-standard-64* Google Cloud instance with an Intel Xeon Gold 6268 @ 2.8 GHz processor and 256 GB of memory. For all our benchmarking, we employed parameters with 128-bits of security according to the LWE-estimator from HElib [16,2]. See our list of parameters in Table S1.

Parameter	**SQUiD**
$\mathfrak{m}$	99,928
t	99,929
$\mathfrak{n}$	49,960
log Q	967
λ	128
Slot size (ℓ)	49,960
Max # of patients	2 billion
Minimum server memory (GB)	60

Table S1: *Table showing the HE parameters for SQUiD where $\mathfrak{m}$ is the dimension of the cyclotomic ring, t is the plaintext modulus, $\mathfrak{n}$ is the polynomial degree, $\log Q$ is the bit length of the ciphertext modulus, λ is the security parameter, slot size is the maximum number of patients that can be stored in a single ciphertext for a single attribute, Max # of patients is the maximum number of patients that can be stored in the database, and Minimum server memory gives a recommendation for the memory requirement of the server to run SQUiD.*

API parameters for queries. ***Count***: Our simplest query counts the number of patients who pass a filter. This filter can be either conjunctive (And) or disjunctive (Or) and consist of only equality causes. For example in Fig. 4A, we have the query "COUNT WHERE $\text{SNP}_2 == 1$ AND sex = 1" which will count

Query	Parameters	Description
Count	Filter: array of pairs of strings and ints Conjunctive: boolean	Filter is [("rs7903146",2),("T2D",1)] Conjunctive is true Returns the number of patients in the database that have both alleles of the rs7903146 SNP and T2D.
MAF	Filter: array of pairs of strings and ints Conjunctive: boolean Target SNP: string	Filter is [("T2D",1)] Conjunctive is true Target SNP is "rs7903146" Returns the MAF for the rs7903146 SNP of patients in the database that have T2D.
PRS	PRS: array of pairs of strings and ints	Parameters are [("rs12562034", 4), ("rs11240779", 10), ("rs4970383", -4)] Returns the PRS score for each patient by summing up the multiplication of the rs12562034 SNP by 4, the rs11240779 SNP by 10, and the rs4970383 SNP by -4.
Similarity	Target patient: ciphertext Threshold: int Disease column: string	Target patient is MO8v3J7kJt (Encrypted) Threshold is 10 Disease column is T2D Returns the number of patients with and without T2D who are similar to the target patient by a threshold of 10 using a L_2 similarity score.

Table S2: *List of parameters for the count, MAF, PRS, and similarity queries with examples.*

the number of female patients where the second SNP is 1. We compute this query by first computing a predicate for each patient if they pass the filter. Since homomorphic encryption only supports addition and multiplication, we compute the filter consisting of AND gates, OR gates, and equality function using polynomial approximations. The domain of the inputs to all these functions is limited to (0,1,2) for SNPs and (0,1) for all other variables in our database. Thus, the degree of these polynomials is small which keeps SQUiD performant.

MAF: Our minor allele frequency (MAF) query computes the MAF of a target SNP on a filtered cohort of patients filtered using a similar filtering technique from the count query. Our MAF queries return two values, the allele count and the number of filtered patients times two. The researcher has to finalize the MAF computation by dividing the returned allele count by the returned number of filtered patients times two. We move the division operation to the research because division is an expensive operation to perform homomorphically.

PRS: Our polygenic risk score (PRS) query computes a PRS given a set of SNPs and a set of effect sizes. A PRS score is calculated with a linear combination of the SNP and the effect sizes which we can naturally compute using homomorphic encryption. The effect sizes, which are often represented as floating point numbers, are scaled to integers within SQUiD as SQUiD only supports integer inputs. The resulting scores are scaled down accordingly.

Similarity: The similarity query counts the number of patients with and without a disease from a cohort consisting of patients similar to the target query patient. This target patient is encrypted with the data owner's public key before it is sent to the cloud to protect the target patient data. To compute the

similarity query, we first compute a similarity cohort by scoring the similarity of the target patient with every patient in the database. We score patients based on their squared Euclidean distance (squared L_2 distance) from the target patient. If these scores are beneath a predetermined threshold, the patient is considered similar to the target patient and added to the similarity cohort. Next, we count the number of patients with and without a disease in this cohort.

The BGV scheme. Brakerski-Gentry-Vaikuntanathan (BGV) [6] is a well-known lattice-based homomorphic encryption scheme that allows for computations over encrypted data. Its lattice-based structure further provides reasonable quantum resistance. Realizing computations as sequences of additions and multiplications, BGV provides a large computation capacity with reasonable parameter choices. Furthermore, it supports computations in a SIMD manner, which significantly enhances the amortized performance.

Below we describe the basic procedures in BGV, the realization of public key-switching and its noise control. Mathematical and cryptographic notations are listed in Table S3.

Mathematical Basics:	
t	plaintext modulus
Q	ciphertext modulus
$\mathbb{Z}$	set of integers
$\mathbb{Z}_t$	the ring of integers modulo t, *i.e.*, $\mathbb{Z}_t = \mathbb{Z}/t\mathbb{Z}$
$\mathbb{Z}_Q$	the ring of integers modulo Q, *i.e.*, $\mathbb{Z}_Q = \mathbb{Z}/Q\mathbb{Z}$
X	variable of polynomial
R	the cyclotomic ring $\mathbb{Z}[X]/\left(\Phi_{\mathfrak{m}}(X)\right)$ where $\Phi_{\mathfrak{m}}(X)$ is the $\mathfrak{m}$-th cyclotomic polynomial. The degree of $\Phi_{\mathfrak{m}}(X)$ is $\mathfrak{n} = \phi(\mathfrak{m})$, where $\phi(\cdot)$ denotes the Euler totient function.
R_t	$\mathbb{Z}_t[X]/\Phi_{\mathfrak{m}}(X)$
R_Q	$\mathbb{Z}_Q[X]/\Phi_{\mathfrak{m}}(X)$
HE Parameters:	
χ_{ter}	uniform ternary distribution with coefficients from $\{-1, 0, 1\}$
χ_{err}	error distribution of scheme, typically a discrete Gaussian distribution unless specified otherwise

Table S3: *Notations of BGV homomorphic encryption scheme*

Key generation, encryption and decryption Sample the secret key sk from χ_{ter}, and the public key is computed as follows

$$\mathsf{pk} = \left([a \cdot \mathsf{sk} + te]_Q, -a\right) \in R_Q^2,$$

where $a \leftarrow u_Q$ and $e \leftarrow \chi_{\mathsf{err}}$. Anyone can see the public key, but this leaks no information on the secret key sk, as guaranteed by the hardness of the ring learning-with-errors (RLWE) problem.

With the public key, anyone can generate a ciphertext for a plaintext m by computing

$$\mathsf{ct} = \mathbf{Enc}_{\mathsf{pk}}(m) = \left([[m]_t + u \cdot \mathsf{pk}_0 + te_0]_Q, [u \cdot \mathsf{pk}_1 + te_1]_Q\right),$$

where $u \leftarrow \chi_{\mathsf{ter}}$ and $e_0, e_1 \leftarrow \chi_{\mathsf{err}}$. Due to the randomness in encryption, ciphertexts of the same plaintext m are always not identical, reflecting the IND-CPA security which avoids the search pattern leakage.

A message m and its ciphertext $(\mathsf{ct}_0, \mathsf{ct}_1)$ are related via

$$\mathsf{ct}_0 + \mathsf{ct}_1 \cdot \mathsf{sk} = [m]_t + tv \pmod{Q},$$

where $v = u \cdot e + e_1 \cdot \mathsf{sk} + e_0$ is the noise term.

Only those knowing the secret key sk will be able to decrypt. Decrypting a ciphertext $ct = (ct_0, ct_1)$ is to calculate

$$[\mathsf{ct}_0 + \mathsf{ct}_1 \cdot \mathsf{sk} \pmod Q)]_t \,,$$

and it decrypts to the correct message m if the noise term v is lower than $\lfloor Q/t \rfloor$.

Public key-switching The key-switching technique is used to switch the secret key of a given ciphertext without decryption. Given a ciphertext ct that encrypts a message m under secret key sk, the goal is to obtain a new ciphertext that encrypts the same message m under another secret key sk^*.

To compute this, the algorithm requires an additional ciphertext that encrypts the secret key sk under sk^*, which is called the key-switching key $\mathsf{ksk}_{(\mathsf{sk}\to\mathsf{sk}^*)}$. In the BGV scheme, this key is derived from sk^*, but we use an alternative derivation with pk^*, the corresponding public key of sk^*, as follows:

$$\mathsf{ksk} = \Big([\mathsf{sk} + u^* \cdot \mathsf{pk}_0^* + te_0^*]_Q \,, [u^* \cdot \mathsf{pk}_1^* + te_1^*]_Q \Big) \in R_Q^2,$$

where $\mathsf{pk}^* = (\mathsf{pk}_0^*, \mathsf{pk}_1^*)$, $u^* \leftarrow \chi_{\mathsf{key}}$ and $e_0^*, e_1^* \leftarrow \chi_{\mathsf{err}}$. In other words, ksk is just $\mathbf{Enc}_{\mathsf{pk}^*}(\mathsf{sk})$, i.e. the encryption of sk under pk^*.

With ksk and $(\mathsf{ct}_0, \mathsf{ct}_1)$ which decrypt to m under sk, we can construct $(\mathsf{ct}_0^*, \mathsf{ct}_1^*)$ which also decrypt m but under sk^*. Precisely, we construct

$$(\mathsf{ct}_0^*, \mathsf{ct}_1^*) = \Big([\mathsf{ct}_0 + \mathsf{ct}_1 \cdot \mathsf{ksk}_0]_Q \,, [\mathsf{ct}_1 \cdot \mathsf{ksk}_1]_Q \Big) \,.$$

Therefore,

$$\begin{aligned}
\mathsf{ct}_0^* + \mathsf{ct}_1^* \cdot \mathsf{sk}^* &= \mathsf{ct}_0 + \mathsf{ct}_1 \cdot \mathbf{Enc}_{\mathsf{pk}^*}(\mathsf{sk})_0 + ct_1 \cdot \mathbf{Enc}_{\mathsf{pk}^*}(\mathsf{sk})_1 \cdot \mathsf{sk}^* \\
&= \mathsf{ct}_0 + \mathsf{ct}_1 \cdot (\mathbf{Enc}_{\mathsf{pk}^*}(\mathsf{sk})_0 + \cdot \mathbf{Enc}_{\mathsf{pk}^*}(\mathsf{sk})_1 \cdot \mathsf{sk}^*) \\
&= \mathsf{ct}_0 + \mathsf{ct}_1 \cdot (\mathsf{sk} + tv^*) \\
&= m + t(v + \mathsf{ct}_1 \cdot v^*) \mod Q,
\end{aligned}$$

where $v^* = u^* \cdot e^* + e_1^* \cdot \mathsf{sk}^* + e_0^*$, verifying $(\mathsf{ct}_0^*, \mathsf{ct}_1^*)$ which also decrypts m but under sk^*.

Reducing noise during public key-switching The public key-switching above has noise component $ct_1 \cdot v^*$, whose size can be further reduced by two methods: coefficient digit decomposition [7], and a temporary enlargement of ciphertext modulus [14]. We combine the two methods in an approach similar to the Section 4.3 of [15], where the ciphertext modulus $Q = \prod_{j=1}^{l} D_j$ is decomposed into l coprime and odd digits, and an expansion factor P, which is odd and coprime to Q, is introduced.

As such, the key-switching key $\mathsf{ksk}_{(\mathsf{sk}\to\mathsf{sk}^*)}$ is no longer a ciphertext with two components, but a matrix of dimension $2 \times l$. For $1 \leq j \leq l$, the j-th column of this matrix is $\mathbf{Enc}_{\mathsf{pk}^*}(P\check{D}_j\mathsf{sk})$ with respect to an enlarged ciphertext modulus PQ, where $\check{D}_j$ stands for the product $D_1 \cdots D_{j-1}$.

Storage costs of ciphertexts and public key-switching keys The size of a ciphertext is dependent on the HE parameters used. With the parameters used in SQUiD, a ciphertext is a pair of polynomials of degree $\mathfrak{n} = 49,960$ with ciphertext modulus $Q \approx 967$ bits. The theoretical minimum storage would be

$$\begin{aligned}
12 \text{ MB} \approx\ & 2 \text{ polynomials per ciphertext} \\
& \times 49{,}961 \text{ coefficients per polynomial} \\
& \times 967 \text{ bits per coefficient.}
\end{aligned}$$

In practice, we measured the storage of a single ciphertext to be 13 MB.

With the coefficient digit decomposition and modulus enlargement techniques, the ksk is a $2 \times l$ matrix of polynomials with enlarged ciphertext modulus PQ. In practice, we set $l = 3$ and $PQ \approx 1,291$ bits. Thus, the theoretical minimum storage of a ksk would be

$$\begin{aligned}48 \text{ MB} \approx\ & 2 \times 3 \text{ polynomials per } ksk \\ & \times 49{,}961 \text{ coefficients per polynomial} \\ & \times 1{,}291 \text{ bits per coefficient.}\end{aligned}$$

In practice, we measured it to be 55 MB.

Homomorphic operations Adding two ciphertexts $\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$ and $\mathsf{ct}' = (\mathsf{ct}'_0, \mathsf{ct}'_1)$ that encrypt m and m' with a same key sk respectively gives

$$(\mathsf{ct}_0 + \mathsf{ct}'_0) + (\mathsf{ct}_1 + \mathsf{ct}'_1) \cdot \mathsf{sk} = [m + m']_t + t(v + v' + u) \pmod Q,$$

so the ciphertext $([\mathsf{ct}_0 + \mathsf{ct}'_0]_Q, [\mathsf{ct}_1 + \mathsf{ct}'_1]_Q)$ is an encryption of $m + m' \pmod t$ under almost additive noise.
Multiplication of two ciphertexts is more complex, which involves taking the tensor product of two ciphertexts as polynomial vectors to obtain $(\mathsf{ct}_0\mathsf{ct}'_0, \mathsf{ct}_0\mathsf{ct}'_1 + \mathsf{ct}_1\mathsf{ct}'_0, \mathsf{ct}_1\mathsf{ct}'_1)$. One can check that

$$\mathsf{ct}_0\mathsf{ct}'_0 + (\mathsf{ct}_0\mathsf{ct}'_1 + \mathsf{ct}_1\mathsf{ct}'_0) \cdot \mathsf{sk} + \mathsf{ct}_1\mathsf{ct}'_1 \cdot \mathsf{sk}^2 = m \cdot m' + tv_0 \pmod Q,$$

where $v_0 = m \cdot v' + m' \cdot v + v \cdot v'$.

Here, an additional step is needed for the term with sk^2. If we consider $(0, \mathsf{ct}_1\mathsf{ct}'_1)$ as a ciphertext with secret key sk^2, then performing the key switching procedure with the key $\mathsf{ksk}_{\mathsf{sk}^2 \to \mathsf{sk}}$ gives a ciphertext $(\mathsf{ct}''_0, \mathsf{ct}''_1)$. The noise control in this step is similar to the previous section. The final output of the homomorphic multiplication is $(\mathsf{ct}_0\mathsf{ct}'_0 + \mathsf{ct}''_0, \mathsf{ct}_0\mathsf{ct}'_1 + \mathsf{ct}_1\mathsf{ct}'_0 + \mathsf{ct}''_1)$.

SIMD batching The BGV scheme supports homomorphic operations on multiple plaintext slots simultaneously. This follows from the fact that the cyclotomic polynomial $\Phi_{\mathfrak{m}}(X)$ of degree $\mathfrak{n}$ splits modulo t into ℓ irreducible factors of same degree $\mathfrak{n}/\ell$, *i.e.*, $\Phi_{\mathfrak{m}}(X) = \prod_{i=1}^{\ell} F_i(X)$. Leveraging the Chinese Remainder Theorem (CRT), the following ring isomorphism

$$R_t \cong \prod_{i=1}^{\ell} \mathbb{Z}_t[X]/\big(F_i(X)\big)$$

is established, which enables the encoding of ℓ messages $\{z_1 \dots, z_\ell\} \in \prod_{i=1}^{\ell} \mathbb{Z}_t[X]/\big(F_i(X)\big)$ into a single plaintext in R_t. Typically, each of the ℓ messages is called a *slot*, and altogether they are regarded as a length-l vector. Since computations over a ciphertext are performed on all packed values, BGV provides efficient computations in an amortized manner.

Homomorphic rotation BGV supports an additional operation called *rotation*, which permutes plaintext slots circularly. Specifically, let ct be an encryption of a plaintext vector $\mathbf{z} = (z_1, z_2, \dots, z_\ell)$. Performing a (right) rotation on ct by v results in a new ciphertext $\mathsf{ct}_{\mathsf{rot}}$ encrypting the plaintext vector $\mathbf{z}_{\mathsf{rot}} = (z_{v+1}, z_{v+2}, \dots, z_\ell, z_1, \dots, z_v)$ under the same secret key.

The homomorphic rotation consists of two key components: automorphism and key switching. We refer interested readers to Section 3 of [15] for the general hypercube structures of rotations in BGV.

Polynomial evaluation using the Paterson-Stockmeyer method Polynomial evaluations require numerous binary operations including additions, scalar multiplications (where one operand is a constant), and non-scalar multiplications. The Paterson-Stockmeyer method [20] uses fewer non-scalar multiplications, such that a degree-d polynomial is evaluated using $\mathcal{O}(\sqrt{d})$ non-scalar multiplications.

In the homomorphic evaluation of polynomials, non-scalar multiplications are translated to ciphertext-ciphertext multiplications, whose evaluation costs are much more expensive than the other two. According to the benchmark in [13], it is around 160× and 15× the cost of homomorphic evaluations of additions and scalar multiplications, respectively. Therefore, the Paterson-Stockmeyer method has been widely used for polynomial evaluations in homomorphic encryption [8,10,11,17].

Below we follow [11] to sketch the evaluation of a degree-d polynomial $f(x) = \sum_{i=0}^{d} a_i x^i$ using the Paterson-Stockmeyer method. Assume there exist integers L and H such that $d = LH - 1$ and $L \approx \sqrt{2(B+1)}$. Then the polynomial can be rewritten into

$$f(x) = \sum_{i=0}^{H-1} (\sum_{j=0}^{L-1} a_{iL+j} \cdot x^j) \cdot x^{iL}.$$

Therefore, the "low powers" $\{x, x^2, \ldots, x^{L-1}\}$ can be computed with $L-2$ non-scalar multiplications, whose linear combinations give the inner sum. The "high powers" $\{x^L, x^{2L}, \ldots, x^{(H-1)L}\}$ are then computed with $H-1$ non-scalar multiplications, whose subsequent products with the inner sum require another $H-1$ non-scalar multiplications. In total, the procedure requires

$$L - 2 + 2(H-1) = L + 2H - 4$$

non-scalar multiplications, which is minimal and achieves $\mathcal{O}(\sqrt{B})$ when $L \approx \sqrt{2(B+1)}$.

Comparison of SQUiD's MAF Calculation with Existing Methods In the MAF protocol by Kim and Lauter [18], the MAF for SNP j is computed as follows:

$$\mathsf{MAF}(j) = \frac{\min(m_j, 2r - m_j)}{2r} \quad \text{with} \quad m_j = \sum_{i=1}^{r} m_{i,j},$$

where r represents the number of patients in the database, and $m_{i,j}$ denotes the genotype of patient i at SNP j.

In SQUiD, a cohort is initially created before the MAF computation. The membership of the cohort is defined by a predicate vector p, which is 1 for patients in the cohort and 0 otherwise. The key distinction in the MAF computation between SQUiD and [18] lies in the need for multiplication of each $m_{i,j}$ term within the summation by the corresponding predicate, ensuring the inclusion of only those patients who are part of the cohort. Additionally, the total number of patients r in the denominator is a constant in [18], but in SQUID it varies for different cohort sizes and needs to be computed as the sum of predicates. All other parts of the MAF computation are the same in both SQUiD and [18]. That is, a SQUiD MAF query without a filter has the same computation and result as [18]. Furthermore, in both SQUiD and [18], the division and minimum operations are executed in plaintext. In summary, the total runtime and accuracy of the SQUiD MAF calculation and that of [18] are expected to be exactly the same.

Continuous phenotype values. SQUiD supports continuous phenotype values such as weight, blood pressure, heart rate, etc. However, as SQUiD exclusively processes integer data inputs, these values need to be discretized into integers by scaling the values. This discretization occurs during the data encryption by the data owner before transmitting it to the cloud. Once in the cloud, SQUiD effectively filters these

now integer values for counting and MAF queries using range filters. Range filters are set by an upper and lower bound and are computed using the same comparisons thresholds from the similarity query.

We benchmarked the range query performance in Additional file 1: Fig S12. We found that it took approximately 51 minutes to compute a count query with a range filter on 49,960 patients and 58 minutes to compute a MAF query with the same parameters.

References

1. Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.
2. Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.
3. Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part I*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, 2011.
4. Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM*, 50(4):506–519, 2003.
5. Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.
6. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.
7. Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Advances in Cryptology–CRYPTO 2011: 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings 31*, pages 505–524. Springer, 2011.
8. Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 34–54. Springer, 2019.
9. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 409–437. Springer, 2017.
10. Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. Efficient homomorphic comparison methods with optimal complexity. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part II*, volume 12492 of *Lecture Notes in Computer Science*, pages 221–256. Springer, 2020.
11. Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. Labeled PSI from homomorphic encryption with reduced computation and communication. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 1135–1150. ACM, 2021.
12. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 144, 2012.

13. Robin Geelen, Michiel Van Beirendonck, Hilder V. L. Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios D. Dimou, Ingrid Verbauwhede, Frederik Vercauteren, and David W. Archer. BASALISC: programmable hardware accelerator for BGV fully homomorphic encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(4):32–57, 2023.
14. Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the aes circuit. In *Advances in Cryptology–CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 850–867. Springer, 2012.
15. Shai Halevi and Victor Shoup. Design and implementation of helib: a homomorphic encryption library. Cryptology ePrint Archive, Paper 2020/1481, 2020. https://eprint.iacr.org/2020/1481.
16. IBM. Helib: An implementation of homomorphic encryption (2.0.0). https://github.com/homenc/HElib, January 2021.
17. Ilia Iliashenko and Vincent Zucca. Faster homomorphic comparison operations for bgv and bfv. *Proceedings on Privacy Enhancing Technologies*, 2021(3):246–264, 2021.
18. Miran Kim and Kristin Lauter. Private genome analysis through homomorphic encryption. *BMC Medical Informatics and Decision Making*, 15(5):S3, Dec 2015.
19. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)*, 60(6):1–35, 2013.
20. Mike Paterson and Larry Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM J. Comput.*, 2:60–66, 03 1973.
21. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.
22. Oded Regev. The learning with errors problem (invited survey). In *Proceedings of the 25th Annual IEEE Conference on Computational Complexity, CCC 2010, Cambridge, Massachusetts, USA, June 9-12, 2010*, pages 191–204. IEEE Computer Society, 2010.
23. Ahmed Salem, Pascal Berrang, Mathias Humbert, and Michael Backes. Privacy-preserving similar patient queries for combined biomedical data. *Proc. Priv. Enhancing Technol.*, 2019(1):47–67, January 2019.

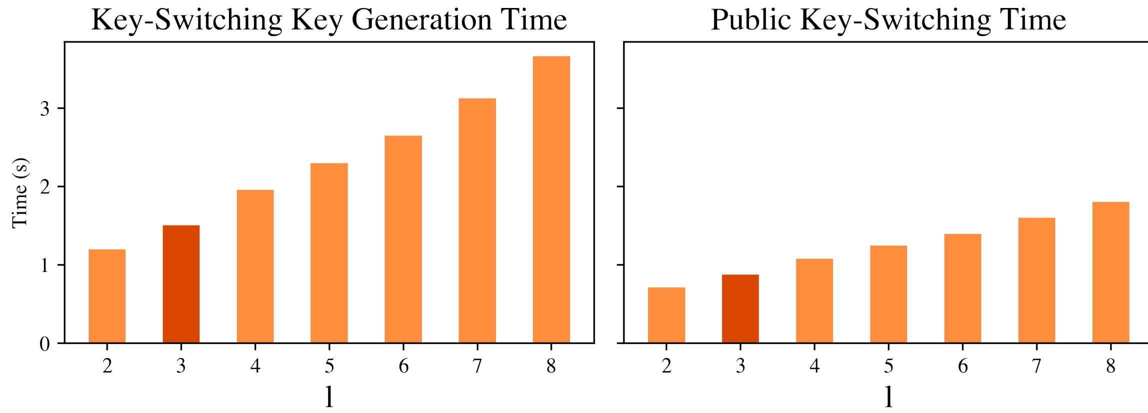

Fig. S1: *The runtime for key-switching key generation and key-switching by the number of digits used in the decomposition (l). Digit decomposition is a technique to reduce the error growth in ciphertexts. The more we decompose the ciphertext and key-switching key, the more secure our system becomes and the less error growth the ciphertext experiences [15]. We set $l = 3$, as is commonly chosen. It takes 1.5 seconds to generate a key-switching key when the digits are decomposed into three parts.*

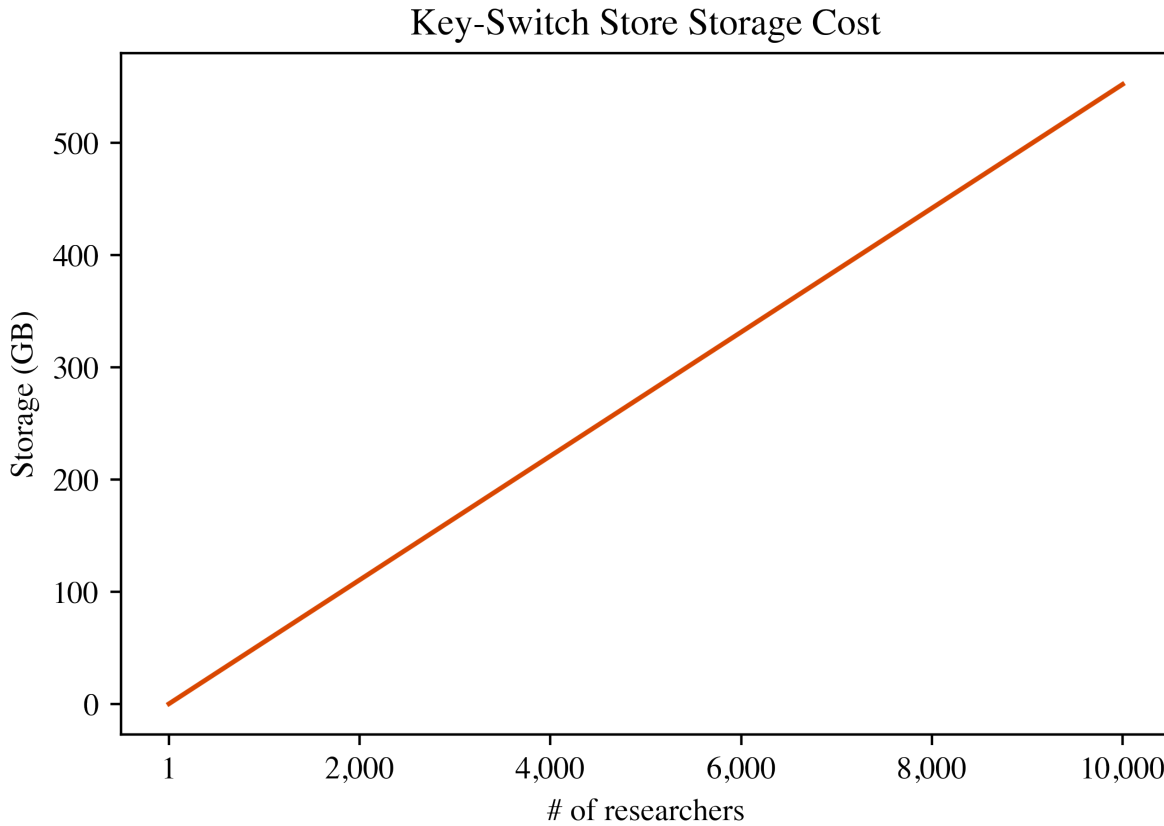

Fig. S2: *The additional storage space for the public key-switching key store (with $l = 3$) by the number of authorized researchers in the store. Since each key-switching key is a single ciphertext, the storage required remains minimal at approximately 55 gigabytes (GB) for 1,000 researchers.*

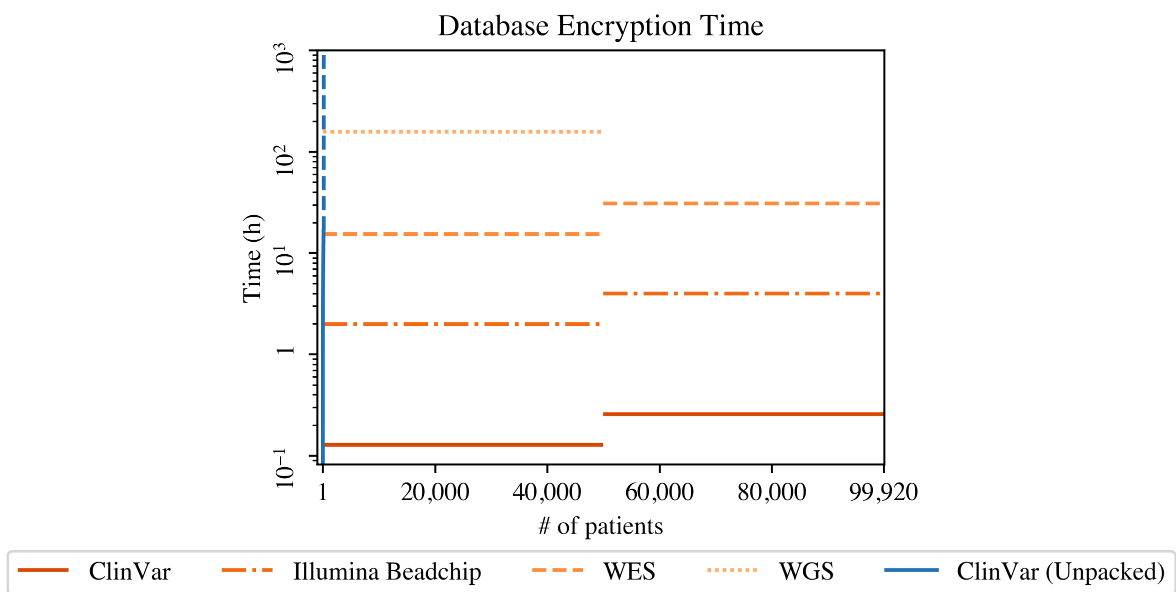

Fig. S3: *The time to encrypt databases with various SNP sets by the number of patients in the database. We measured the setup time by encrypting the various databases using 60 threads running simultaneously. We compared the setup of SQUiD for the ClinVar SNP set to the setup time of an unpacked HE solution (blue) for the ClinVar SNP set. The ClinVar (Unpacked) line was extrapolated for databases that took more than a day to generate.*

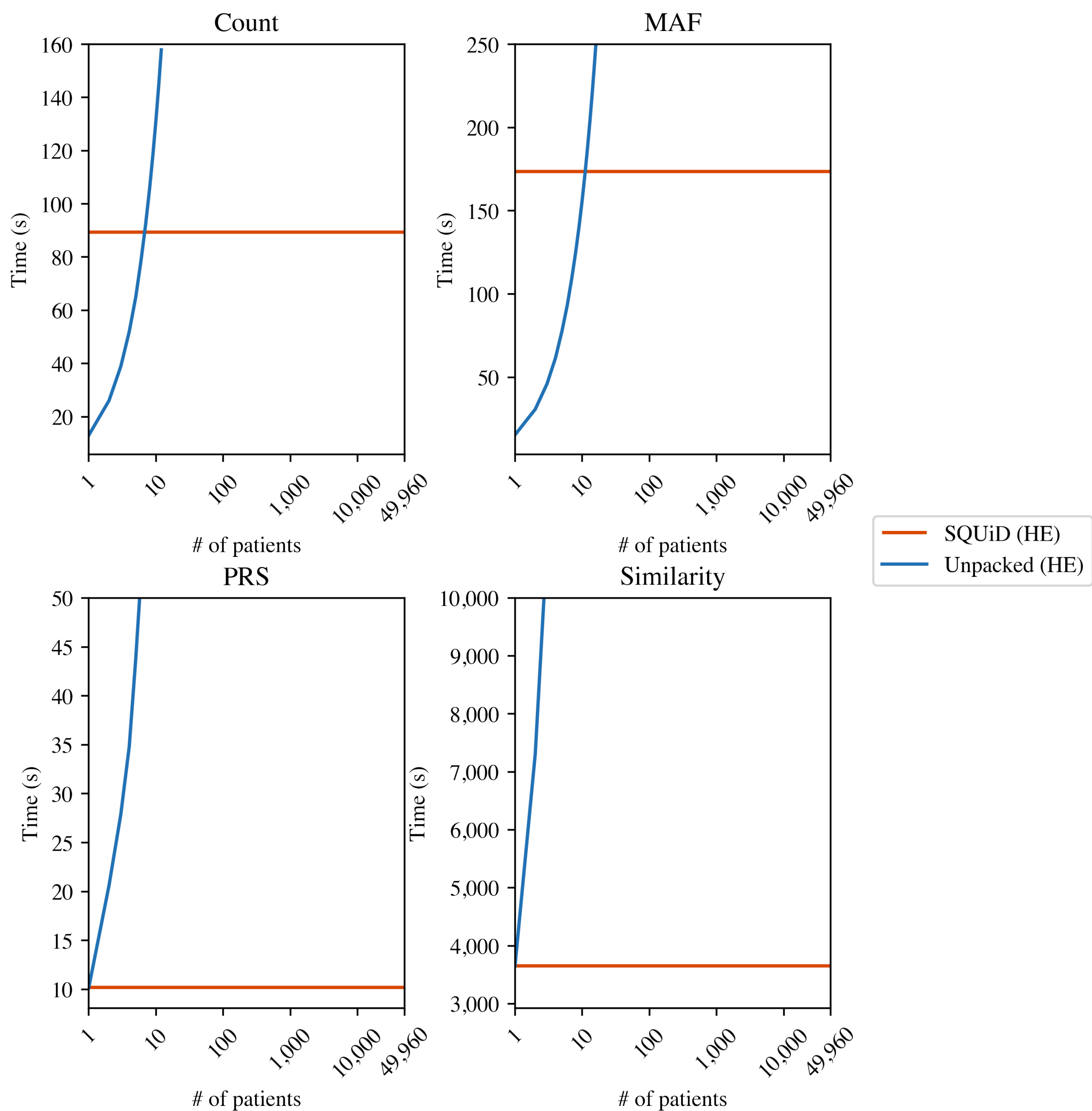

Fig. S4: *The unpacked and packed query performance for the count, MAF, PRS, and similarity queries on databases with varying numbers of patients (ranging from 1 patient to 49,960 patients with 1,024 SNPs). The similarity and PRS queries were tested with 1,024 effect sizes and SNPs, respectively, and the count and MAF queries were tested with 2 filters.*

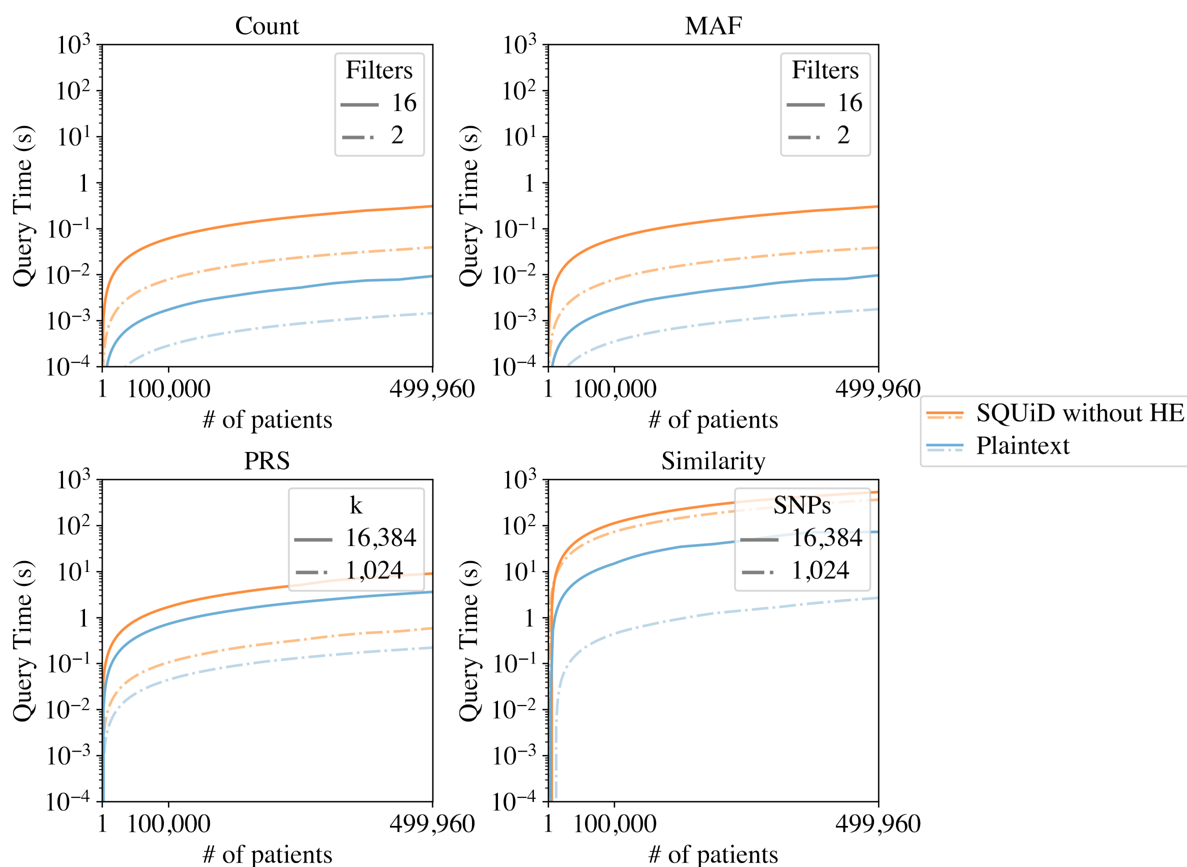

Fig. S5: *The query time for SQUiD without HE and plaintext. SQUiD without HE implements the queries from SQUiD by translating HE library functionalities to their plaintext counterparts and then using these functionalities in the same way as SQUiD. We ran the count and MAF queries with 2 filters, the PRS query with 1,024 effect SNPs (k), and the similarity query with 1,024 SNPs.*

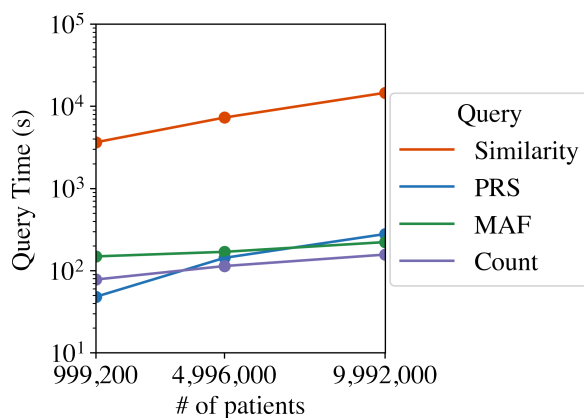

Fig. S6: *The time to perform count, MAF, PRS and similarity queries on databases with an increasing number of patients. The count and MAF queries were run with 2 filters. The PRS query had 1,024 effect SNPs, and the similarity query had 1,024 SNPs. Each query was executed in parallel with 50 threads.*

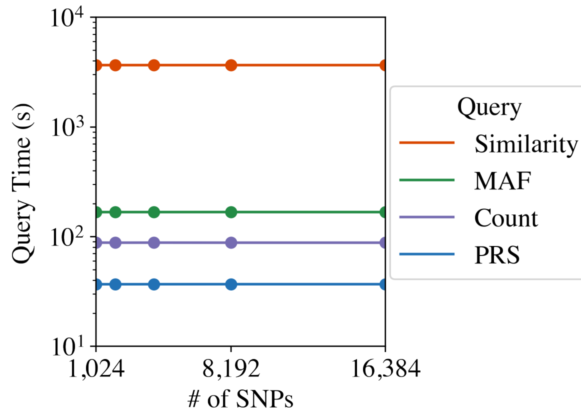

Fig. S7: *The time to perform a count query with 2 filters, MAF query with 2 filters, PRS with 1,024 effect SNPs, and a similarity queries with 1,024 SNPs on databases with increasing number of SNPs.*

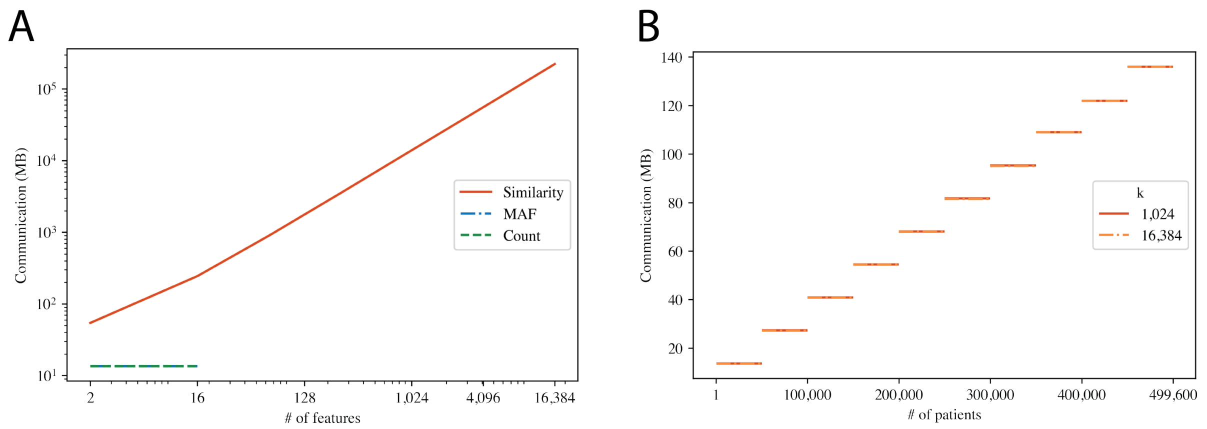

Fig. S8: *The communication cost for all queries by the number of features in the query or the number of patients in the database. We benchmark our communication cost by measuring the end-to-end communication of a single query. Each query needs one communication round thus the total communication is the cost of receiving a query from a client and sending back the computation result. We separated the communication performance into two plots because the scaling for our queries depend on different factors. (A) We show the communication costs of the count, MAF, and similarity queries by the number of features (filters for the count and MAF query, and SNPs for the similarity query). (B) We show the communication of the PRS query by the number of patients in the database.*

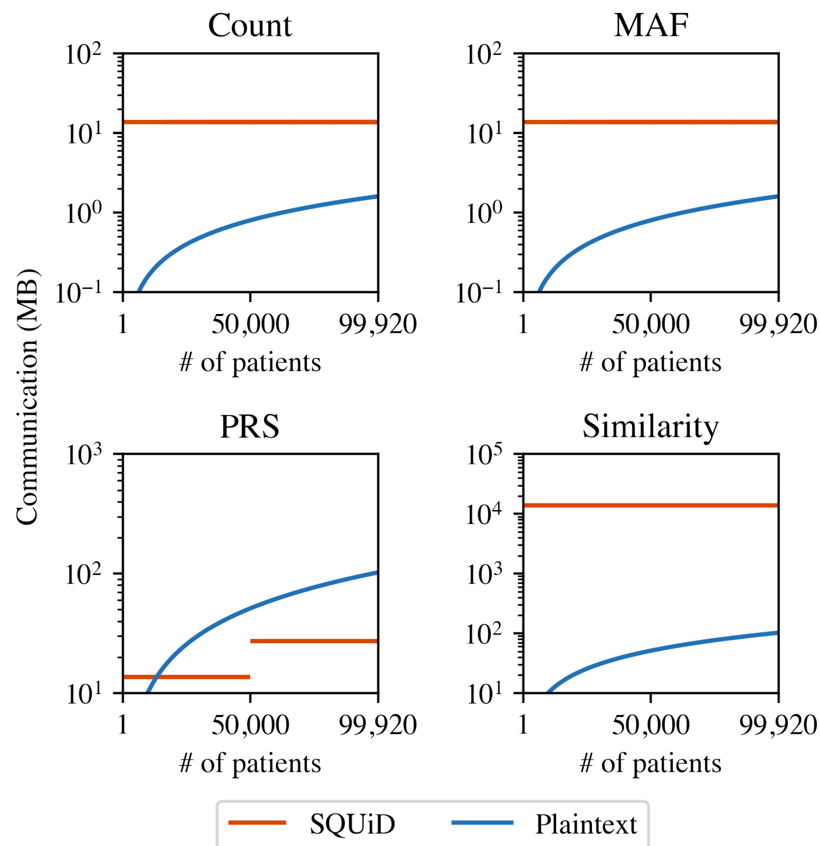

Fig. S9: *The communication cost for all queries and a plaintext solutions by the number of patients in the database. The plaintext database keeps the data encrypted at rest, packing 16 one byte SNPs into a single encrypted 128-bit AES block. When the database receives a query, it sends the encrypted data necessary to compute the query. Thus, the client has to decrypt and compute the query themselves. We benchmarked for all four types of queries with 16 filters for the count and MAF queries, 1,024 effect SNPs for the PRS query, and the 1,024 SNPs for the similarity query. Compared to SQUiD, the plaintext communication always scales linearly with the number of patients while this is only true for SQUiD for PRS queries (since a PRS score for each patient needs to be returned).*

```
A > ./bin/squid
  Welcome to SQUiD!
  --- Setup ---
  Config server address, port, and API key: ./bin/squid config <address> <port> <api_key>
  Pull context from server: ./bin/squid getContext
  Generate own context: ./bin/squid genContext
  Generate public / secret key: ./bin/squid genKeys
  Authorize yourself to the server (by generating key-switching key): ./bin/squid authorize

  --- Query ---
  Query: ./bin/squid <option> [query_string]

  --- Helper ---
  Decrypt query results (for queries not automatically decrypted): ./bin/squid decrypt <file>
B > ./bin/squid config localhost 8081 nNCHuSdBWZsDJNFOJqUWDAUibEvVcVniRqbiIoM
  11:57:49: Set config
C > ./bin/squid getContext
  11:57:57: Requesting context
  11:57:57: Received context
D > ./bin/squid genKeys
  11:58:05: Loaded in context
  11:58:05: Generated secret key
  11:58:05: Generated public key
  11:58:05: Wrote secret key to file
  11:58:05: Wrote Public Key to File
E > ./bin/squid authorize
  11:58:17: Sending public key
  11:58:18: Authorization successful
F > ./bin/squid count "[(1,1)]" 1
  11:58:33: Counting Query with:
  11:58:33: filter: [(1,1)]
  11:58:33: conjunctive: And
  11:58:33: Count:4
G > ./bin/squid MAF
  Not enough parameters for the MAF query [filter] [conjunctive] [target snp]
H > ./bin/squid MAF "[(1,1)]" 1 2
  11:59:32: MAF Query with:
  11:59:32: filter: [(1,1)]
  11:59:32: conjunctive: And
  11:59:32: target: 2
  11:59:32: MAF: 0.25
  >
```

Fig. S10: *Screenshot of SQUiD command line interface (CLI).* ***(A)*** *Welcome help message about showing core SQUiD functionalities.* ***(B)*** *setConfig call sets the address, port, and API key used to communicate with the server for all successive calls.* ***(C)*** *getContext call that pulls the context from the server to ensure the encryption schemes are synced locally and on the server.* ***(DC*** *genKeys call creates a public and private key for the user.* ***(E)*** *authorize call sends the public key to the server for authorization. The server generates a key-switching key which will be used to re-encrypt all query results sent back to the user to be encrypted under the user's public key.* ***(F)*** *count query call which counts the number of patients in the database with a first SNP that has value 1.* ***(G)*** *failed query call that shows what parameters should be supplied for a correct query call.* ***(H)*** *MAF query call that has the correct parameters.*

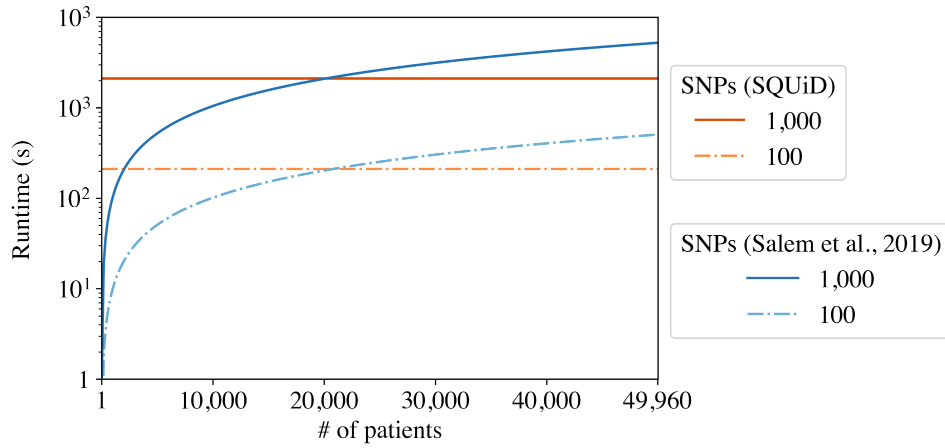

Fig. S11: *The runtime of the L_2 similarity score computation for SQUiD and [23] (Salem et al., 2019) for 100 SNPs and 1,000 SNPs. For score computation with fewer than 20,000 patients, [23] exhibits faster performance for both 100 and 1,000 SNPs. However, as the patient dataset scales up, SQUiD consistently outperforms [23], demonstrating its superior efficiency in handling larger datasets.*

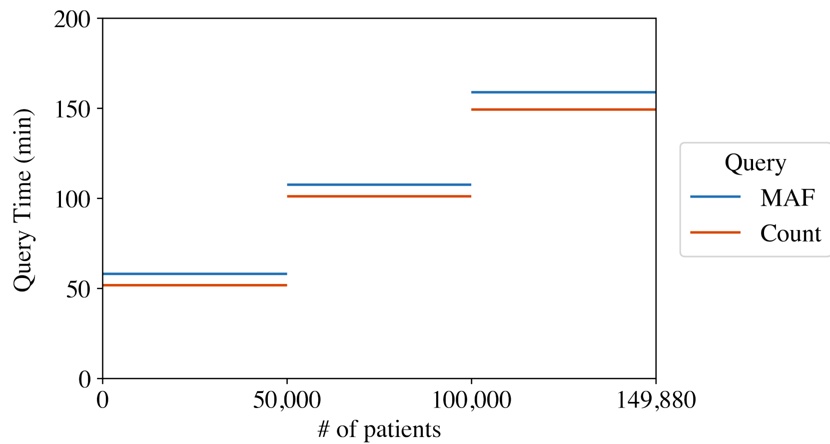

Fig. S12: *The query time for the count and MAF query with a range filter by the number of patients in the database. Each query only had one range filter.*