Supplementary Note 1: Our Protocol Setup

There are four parties in our GWAS protocol: study participants (SP), two main computing parties (CP_1 and CP_2), and an auxiliary computing party (CP_0). The auxiliary computing party CP_0 only participates in a *precomputation* (or preprocesisng) phase of the protocol, and in particular, does not participate in the main protocol execution. For simplicity, we take SP to be a single entity that possesses all of the input genomes and phenotypes. As we discuss in Supplementary Note 9, this setting naturally generalizes to the crowdsourcing scenario where SP represents many independent participants, each securely contributing their own genome to the computing parties.

Overview. The overall workflow of our GWAS protocol is as follows. First, using a cryptographic technique called *secret sharing*, the study participants SP share their data (i.e., their genome and associated phenotypes) with CP_1 and CP_2 in such a way that enables computation over the shared data, but does not reveal any information to either CP_1 or CP_2. Next, CP_1 and CP_2 engage in an interactive protocol to perform GWAS over the private inputs without learning anything about the underlying data. During this computation, CP_1 and CP_2 leverage pre-computed data from CP_0 (which is input-agnostic) to greatly speed up the computation. Finally, CP_1 and CP_2 combine their outputs to reconstruct the final GWAS statistics from secret shares and publish the results. We work in the general paradigm of computing on secret-shared data first formalized by Ben-Or et. al [1], and subsequently extended in a number of works [2, 3, 4, 5, 6].

Security model. In this paper, we assume that the protocol participants are semi-honest (i.e., honest-but-curious). In other words, all parties follow the protocol exactly as specified, but at the end of the protocol execution, parties may try to infer additional information about other parties' private inputs based on their view of the protocol execution. Informally, we say that a cryptographic protocol is secure in the semi-honest model if all of the information a party is able to learn from the protocol execution can be expressed as a function of just the party's input to the protocol execution and the output of the computation (i.e., the GWAS results). In Supplementary Note 10, we briefly describe ways of extending our protocol to relax our security assumptions and additionally provide security against malicious adversaries in the online phase of the protocol by applying the techniques from the SPDZ online protocol [3].

Communication model. In our protocol, we assume that every pair of parties communicate over a secure and authenticated channel (e.g., over the TLS protocol). Concretely, this means that if CP_1 sends a message to CP_2 during the protocol execution, only CP_2 can read the message; other parties (such as SP, CP_0, or an eavesdropper on the network) are unable to do so.

Nature Biotechnology: doi:10.1038/nbt.4108

Supplementary Note 2: Secure Multiparty Computation Review

Here, we formally describe a cryptographic framework known as secure multiparty computation (MPC) based on secret sharing, which provides the foundation for our secure GWAS protocol.

2.1 Notation

We begin by introducing some notation that we use throughout this work. For a prime q, we write $\mathbb{Z}_q$ to denote the integers modulo q. For a finite set S, we write $x \xleftarrow{R} S$ to denote that x is sampled uniformly at random from S. Our protocols consist of a sequence of operations performed by multiple parties. We annotate each operation with the relevant party or parties that is responsible for performing the operation (unless the operation applies to all parties). We write a tuple with angle brackets (e.g., $\langle a, b, c, d \rangle$) to represent data that are visible to only a subset of the parties in the protocol. The ordering we adopt is $\langle \mathsf{CP}_1, \mathsf{CP}_2, \mathsf{CP}_0, \mathsf{SP} \rangle$. In other words, these tuples are placeholders that take on different values depending on which party is executing the corresponding line of the protocol. We write $\perp$ to denote an empty slot, and we adopt the following abbreviations for compactness:

$$\langle a, b \rangle \equiv \langle a, b, \perp, \perp \rangle \text{ and } \langle a, b, c \rangle \equiv \langle a, b, c, \perp \rangle .$$

We give a few examples of our notation in the table below:

Notation	Description
$\langle a, b, c, d \rangle$	CP_1 sees a, CP_2 sees b, CP_0 sees c, SP sees d
$\langle \perp, a, \perp, a \rangle$	Both CP_2 and SP see a
$\langle a, \perp, b \rangle$	CP_1 sees a, CP_0 sees b
$\langle a, b \rangle$	CP_1 sees a, CP_2 sees b

2.2 Secret Sharing

Our protocol relies on *secret sharing*—a cryptographic technique that allows two or more parties to collectively represent a private value without having any knowledge about it individually. The underlying secret is reconstructed only when a prescribed set of parties (e.g., all of them) combine their respective shares. In our setting, the study participants first secret share their data to the two computing parties CP_1 and CP_2. The protocol execution consists of an interactive protocol where CP_1 and CP_2 jointly compute over the secret-shared data.

Our protocol relies on a two-party additive secret sharing scheme. A two-party additive sharing of a value $x \in \mathbb{Z}_q$ consists of a pair of values $(r, x - r)$ where $r \xleftarrow{R} \mathbb{Z}_q$ is a uniformly random field element. By construction, each share (either r or $x - r$) individually reveals *no* information about the value x. However, given both shares r and $x - r$, it is possible to recover the secret value x by adding together the two shares (hence the name, additive secret sharing). In this work, we denote the two shares $(r, x - r)$ of a value $x \in \mathbb{Z}_q$ by $([x]_1, [x]_2)$, respectively. Using our tuple notation, a secret sharing $[x]$ of a value $x \in \mathbb{Z}_q$ can be written as

$$[x] := \langle [x]_1, [x]_2 \rangle .$$

In words, $[x]_1$ and $[x]_2$ are shares of x individually owned by CP_1 and CP_2, respectively. Adding the two shares together yields the value x. Since CP_1 and CP_2 each only sees a single share of x, individually, they have no information about the value of x. The procedures for sharing

Nature Biotechnology: doi:10.1038/nbt.4108

and reconstructing a secret are summarized in Protocol 1 and 2. Note that the return value of Protocol 2 is $\langle x, x \rangle$, which in our notation, means that both CP_1 and CP_2 learn the shared value x.

Protocol 1: $[x] \leftarrow \mathsf{ShareSecret}(\langle \perp, \perp, \perp, x \rangle)$ (or $\langle \perp, \perp, x \rangle$)

1: SP (or CP_0): Sample a uniformly random element $r \xleftarrow{R} \mathbb{Z}_q$
2: SP (or CP_0): Send r to CP_1 and $x - r$ to CP_2
3: **return** $\langle r, x - r \rangle$

Protocol 2: $\langle x, x \rangle \leftarrow \mathsf{ReconstructSecret}([x])$

1: CP_1: Send $[x]_1$ to CP_2
2: CP_2: Send $[x]_2$ to CP_1
3: $\mathsf{CP}_1, \mathsf{CP}_2$: Compute $x \leftarrow [x]_1 + [x]_2$
4: **return** $\langle x, x \rangle$

2.3 Computing on Secret-Shared Data

Not only do secret sharing schemes allow data to be shared across multiple parties in a privacy-preserving manner, the parties can exploit the structure of the shared data to additionally compute over the private inputs without learning anything about the underlying inputs [1]. Intuitively, this is akin to manipulating objects while blindfolded. Secure computation of an arbitrary function is typically implemented as a composition of subprotocols that are each defined for primitive operations (e.g., addition and multiplication). Each subprotocol takes secret-shared values as input and outputs the intended result also as secret shares, while ensuring that no information about the private input is leaked in between.

Affine functions over shared values. First, consider the case of adding two secret-shared values $[x]$ and $[y]$ without revealing x and y. This is possible by having each party add their own shares (Protocol 3). Note that the resulting shares $[x]_1 + [y]_1 \in \mathbb{Z}_q$ and $[x]_2 + [y]_2 \in \mathbb{Z}_q$ add up to $x + y \in \mathbb{Z}_q$. As long as the shares of $[x]$ and $[y]$ were constructed independently, the resulting shares contain no information about $x+y$. Similarly, adding or multiplying by a public field element $a \in \mathbb{Z}_q$ (represented as $\langle a, a \rangle$ using our notation) is also possible. We describe these basic subroutines in Protocols 4 and 5.

Protocol 3: $[x + y] \leftarrow \mathsf{Add}([x], [y])$ (also written $[x] + [y]$)

1: **return** $\langle [x]_1 + [y]_1, [x]_2 + [y]_2 \rangle$

Protocol 4: $[x + a] \leftarrow \mathsf{AddPublic}([x], \langle a, a \rangle)$ (also written $[x] + a$ or $a + [x]$)

1: **return** $\langle [x]_1 + a, [x]_2 \rangle$

Protocol 5: $[ax] \leftarrow \mathsf{MultiplyPublic}([x], \langle a, a \rangle)$ (also written $a[x]$)

1: **return** $\langle a[x]_1, a[x]_2 \rangle$

Protocols 3–5 together enable secure computation of any *affine* function over the private input. For notational simplicity, we write affine functions directly in terms of the secret shares $[\cdot]$ without explicitly showing how the basic subroutines are invoked. For instance, we can express the composite protocol for securely evaluating $f(x, y) = ax - y + b$ given shared inputs $[x]$ and $[y]$ and public values a and b as

$$[f(x, y)] \leftarrow a[x] - [y] + b.$$

Multiplication of shared values. Multiplication of secret-shared values is more complicated. We adopt a useful tool known as *Beaver multiplication triples* [7], which are triples of (correlated) random values that can be used for secure multiplication. More precisely, a Beaver multiplication triple is a secret-sharing of a random product: namely, a triple $([a], [b], [c])$ where $a, b \xleftarrow{R} \mathbb{Z}_q$ are random field elements and $c = ab \in \mathbb{Z}_q$. The key observation is that if two parties possess a secret-sharing of a random multiplication triple, it is possible to compute a secret-sharing of an *arbitrary* product with a small amount of communication. In this work, we take a *server-aided* approach [8] to generate the multiplication triples, where we essentially outsource the generation of the multiplication to an auxiliary party (namely, CP_0) during a preprocessing step. The Sharemind framework [9] for multiparty computation uses a similar technique in their share multiplication protocol. Protocol 6 outlines the generation of a multiplication triple, and Protocol 7 shows how it is used to perform secure multiplication on shared values.

Protocol 6: $([a], [b], [c]) \leftarrow \mathsf{GetMultiplicationTriple}()$

1: CP_0: Sample $a, b \xleftarrow{R} \mathbb{Z}_q$ and compute $c \leftarrow ab$
2: $[a] \leftarrow \mathsf{ShareSecret}(\langle \perp, \perp, a \rangle)$ (in parallel with Lines 3 and 4)
3: $[b] \leftarrow \mathsf{ShareSecret}(\langle \perp, \perp, b \rangle)$
4: $[c] \leftarrow \mathsf{ShareSecret}(\langle \perp, \perp, c \rangle)$
5: **return** $([a], [b], [c])$

Protocol 7: $[xy] \leftarrow \mathsf{Multiply}([x], [y])$ (also written $[x][y]$)

1: $([a], [b], [c]) \leftarrow \mathsf{GetMultiplicationTriple}()$
2: $\langle x - a, x - a \rangle \leftarrow \mathsf{ReconstructSecret}([x] - [a])$ (in parallel with Line 3)
3: $\langle y - b, y - b \rangle \leftarrow \mathsf{ReconstructSecret}([y] - [b])$
4: **return** $(x - a)(y - b) + (x - a)[b] + (y - b)[a] + [c]$

Correctness. Correctness of Protocol 7 follows from the fact that

$$\begin{aligned} xy &= ((x - a) + a)((y - b) + b) \\ &= (x - a)(y - b) + (x - a)b + (y - b)a + ab. \end{aligned}$$

Because this expression is affine over the values in a multiplication triple (a, b, and $c = ab$), it can be securely computed given only the secret-shared triple using Protocols 3–5. Note that $x - a$ and $y - b$ can be treated as public since they are revealed to both parties in Lines 2 and 3 of Protocol 7.

Security. Unlike previous subroutines, Protocol 7 is interactive, so each party sees additional information beyond their initial input shares. Thus, its security needs to be analyzed more carefully. Consider the view of CP_1 in Protocol 7. It observes the secret shares of the multiplication triple

($[a]_1$, $[b]_1$, and $[c]_1$) and the data sent by CP_2 in Lines 2 and 3 ($[x]_2 - [a]_2$ and $[y]_2 - [b]_2$). First, note that $[a]_1$, $[b]_1$, and $[c]_1$ are fresh shares of a, b, and c, and thus are independent of a and b. In addition, a and b are each independently random and unknown to CP_1. This means that $[x]_2 - [a]_2 = ([x]_2 + [a]_1) - a$ and $[y]_2 - [b]_2 = ([y]_2 + [b]_1) - b$ are independent and uniformly random. Thus, the view of CP_1 during the protocol execution contain no information about x and y (all the values CP_1 sees are distributed uniformly and independently random). The same argument holds for CP_2. Overall, neither party learns anything about x and y during Protocol 7 that was not previously known.

With the addition of secure multiplication subroutine (Protocol 7), we now have a general-purpose secure computation framework for arbitrary arithmetic circuits (polynomial functions). In the following, we extend our shorthand notation for composite protocols to include multiplication:

$$[a][b] := \text{Secure multiplication of } [a] \text{ and } [b].$$

Supplementary Note 3: Our Generalization of Beaver Multiplication Triples

Multiplication triples are traditionally used to compute a product of two secret-shared values. In many scenarios, the direct application of multiplication triples to more complex computations (where each multiplication operation invokes Protocol 7) leads to highly inefficient protocols, especially when there are a large number of multiplications. Here, we describe how to generalize multiplication triples to facilitate the evaluation of arithmetic circuits (i.e., polynomials) on secret-shared data. Our approach is much more practical for complex computations such as computing GWAS statistics, and thus, may be of independent interest. We begin by giving an outline of our method:

1. **Input blinding:** First, the computing parties CP_1 and CP_2 "blind" their input value $[x]$ (secret-shared) by subtracting from it a secret random value $[r]$ (also secret-shared). Then, the computing parties publish their shares of $[x-r]$ to reveal the blinded value $x-r$. Here, the blinding factor r is randomly chosen by CP_0 and is secret-shared with CP_1 and CP_2. We refer to this procedure as *Beaver partition* (Protocol 8).

2. **Offline computation:** All parties (CP_0, CP_1, and CP_2) compute over the Beaver-partitioned data according to the function specification. This step is non-interactive.

3. **Output reconstruction:** CP_0 secret shares the results of its computation (over the blinding factors) with CP_1 and CP_2. These shares enable CP_1 and CP_2 to reconstruct the desired output without interaction.

Notably, CP_0's task does not depend on any (secret) input value and thus, can be precomputed and secret-shared in advance. This leads to an efficient protocol where CP_1 and CP_2 only needs to communicate in the first step of the computation. The core of the computation is non-interactive.

More formally, let f be the arithmetic circuit (represented as a polynomial) that we want to evaluate:

$$f(x_1,\ldots,x_n) := \sum_{t=1}^{T} c^{(t)} x_1^{p_1^{(t)}} \cdots x_n^{p_n^{(t)}},$$

where the number of monomials T, the coefficients $c^{(t)}$, and the exponents $p_i^{(t)}$ are public, and the input is given as secret shares $[x_1],\ldots,[x_n]$.

Input blinding. In the first step, the inputs are Beaver-partitioned into (i) public values $x_1 - r_1,\ldots,x_n - r_n$ and (ii) hidden (secret-shared) blinding factors $[r_1],\ldots,[r_n]$ (Protocol 8). The computing parties CP_1 and CP_2 receive $x_1-r_1,\ldots,x_n-r_n$ as well as shares of the blinding factors $[r_1],\ldots,[r_n]$. The auxiliary party CP_0 chooses the blinding factors $r_1,\ldots,r_n$.

Protocol 8: $\langle x-r, x-r\rangle, \langle [r]_1,[r]_2,r\rangle \leftarrow \mathsf{BeaverPartition}([x])$

1: CP_0: Sample $r \xleftarrow{\mathrm{R}} \mathbb{Z}_q$
2: $[r] \leftarrow \mathsf{ShareSecret}(\langle \perp, \perp, r\rangle)$
3: $\langle x-r, x-r\rangle \leftarrow \mathsf{ReconstructSecret}([x]-[r])$
4: **return** $\langle x-r, x-r\rangle, \langle [r]_1,[r]_2,r\rangle$

Nature Biotechnology: doi:10.1038/nbt.4108

Next, we re-express the function f as a polynomial in the variables $r_1, \ldots, r_n$ using the substitution $x_i \mapsto r_i + (x_i - r_i)$. Moreover, we treat the values $x_i - r_i$ as a constant for all i (since this value is publicly known to CP_1 and CP_2):

$$\begin{aligned} g(r_1, \ldots, r_n) &:= \sum_{t=1}^{T} c^{(t)} (r_1 + (x_1 - r_1))^{p_1^{(t)}} \cdots (r_n + (x_n - r_n))^{p_n^{(t)}} \\ &= \underbrace{f(x_1 - r_1, \ldots, x_n - r_n)}_{\text{degree } 0} + \underbrace{\sum_{i=1}^{n} \tilde{d}_i r_i}_{\text{degree } 1} + \underbrace{\sum_{t=1}^{\tilde{T}} \tilde{c}^{(t)} r_1^{\tilde{p}_1^{(t)}} \cdots r_n^{\tilde{p}_n^{(t)}}}_{1 < \text{ degree } < \deg(f)} + \underbrace{f(r_1, \ldots, r_n)}_{\text{degree } \deg(f)}, \end{aligned} \tag{1}$$

where we introduce a new set of parameters: coefficients $\tilde{c}^{(t)}$ and $\tilde{d}_i$, exponents $\tilde{p}_i^{(t)}$, and $\tilde{T}$ denoting the number of intermediate-degree terms (between 1 and $\deg(f)$). We separately group degree-one terms and the terms corresponding to $f(r_1, \ldots, r_n)$ and $f(x_1 - r_1, \ldots, x_n - r_n)$ to simplify our subsequent analysis. We enforce that $(\tilde{p}_1^{(t)}, \ldots, \tilde{p}_n^{(t)})$ be unique for each $t = 1, \ldots, \tilde{T}$, and assume that there is a canonical structure for g that is fixed and known to all of the parties. In other words, $\tilde{T}$ and the new exponents $\tilde{p}_i^{(t)}$ are fixed and known to all parties.

Offline computation. In the second step, CP_1 and CP_2 compute the values of $\tilde{c}^{(t)}$, $\tilde{d}_i$, and $f(x_1 - r_1, \ldots, x_n - r_n)$ from the blinded input $x_1 - r_1, \ldots, x_n - r_n$ derived in the first step of the computation. Meanwhile, CP_0 computes

$$R^{(0)} := f(r_1, \ldots, r_n) \text{ and } R^{(t)} := r_1^{\tilde{p}_1^{(t)}} \cdots r_n^{\tilde{p}_n^{(t)}}, \ \forall t \in \{1, \ldots, \tilde{T}\}, \tag{2}$$

which is possible because CP_0 knows the blinding values $r_1, \ldots, r_n$.

Output reconstruction. Finally, CP_0 secret shares $R^{(0)}, \ldots, R^{(\tilde{T})}$ with CP_1 and CP_2. Then, CP_1 and CP_2 compute

$$f(x_1 - r_1, \ldots, x_n - r_n) + \sum_{i=1}^{n} \tilde{d}_i [r_i] + \sum_{t=1}^{\tilde{T}} \tilde{c}^{(t)} [R^{(t)}] + [R^{(0)}], \tag{3}$$

which is $[g(x_1, \ldots, x_n)]$ by Eq. (1). Note this function is affine over the secret inputs $[r_1], \ldots, [r_n]$ and $[R^{(0)}], \ldots, [R^{(\tilde{T})}]$. Therefore, it can be computed non-interactively using Protocols 3–5.

The overall procedure for securely evaluating a polynomial given a secret-shared input is provided in Protocol 9. When $f(x_1, x_2) = x_1 x_2$, this procedure reduces to the standard Beaver multiplication protocol (Protocol 7).

Protocol 9: $[f(x_1, \ldots, x_n)] \leftarrow \mathsf{EvaluatePolynomial}(f, [x_1], \ldots, [x_n])$

Require: $f(x_1, \ldots, x_n) = \sum_{t=1}^{T} c^{(t)} x_1^{p_1^{(t)}} \cdots x_n^{p_n^{(t)}}$ where T, $c^{(t)}$, and $p_i^{(t)}$ are public for all i, t

/* Step 1: Input blinding */
for each i in $\{1, \ldots, n\}$ **do** (in parallel)
    $\langle x_i - r_i, x_i - r_i \rangle, \langle [r_i]_1, [r_i]_2, r_i \rangle \leftarrow \mathsf{BeaverPartition}([x_i])$
end for

/* Step 2: Offline computation */
CP_0, CP_1, CP_2: Rewrite f according to Eq. (1) to obtain $\tilde{T}$ and $\tilde{p}_i^{(t)}$, $\forall i, t$
CP_1, CP_2: Calculate $\tilde{c}^{(t)}$ and $\tilde{d}_i$, $\forall i, t$, and $f(x_1 - r_1, \ldots, x_n - r_n)$
CP_0: Calculate $R^{(0)} \leftarrow f(r_1, \ldots, r_n)$ and $R^{(t)} \leftarrow r_1^{\tilde{p}_1^{(t)}} \cdots r_n^{\tilde{p}_n^{(t)}}$ for $t = 1, \ldots, \tilde{T}$

/* Step 3: Output reconstruction */
for each t in $\{0, \ldots, \tilde{T}\}$ **do** (in parallel)
    $[R^{(t)}] \leftarrow \mathsf{ShareSecret}(\langle \perp, \perp, R^{(t)} \rangle)$
end for
$[y] \leftarrow f(x_1 - r_1, \ldots, x_n - r_n) + \sum_{i=1}^{n} \tilde{d}_i [r_i] + \sum_{t=1}^{\tilde{T}} \tilde{c}^{(t)} [R^{(t)}] + [R^{(0)}]$
return $[y]$

Security. Security of this protocol follows analogously to that for the Beaver multiplication protocol. Namely, the additional data observed by each party in the protocol execution consists entirely of independent random values in $\mathbb{Z}_q$, and thus, do not reveal any information about any secret input. As before, consider the view of each party. In the first step ($\mathsf{BeaverPartition}$), CP_1 sees values $[x]_2 - [r]_2 = ([x]_2 + [r]_1) - r$ and $[r]_1$, which are distributed uniformly and independently over $\mathbb{Z}_q$ (since r and $[r]_1$ are independent, uniformly random values of $\mathbb{Z}_q$). An analogous argument holds for CP_2. The second step is non-interactive. In the final step, CP_1 and CP_2 receive secret shares of CP_0's computation results, which individually are independent and uniform over $\mathbb{Z}_q$. Therefore, the views of both CP_1 and CP_2 in the protocol execution consists of uniformly random values. To conclude the argument, we note that CP_0 does not receive any messages from any other party in Protocol 9 (so its view is trivially independent of any secret inputs).

Circuits with multiple outputs. Our method can be further generalized to arithmetic circuits with more than one output as follows. Suppose the circuit has ℓ output gates. We can represent each output by a polynomial $f_1, \ldots, f_\ell$ on a common set of inputs $x_1, \ldots, x_n$. To avoid redundant computation, we first write out Eq. (1) for each polynomial, and take the union over all intermediate-degree terms to obtain a set $\mathcal{P}$ of exponents $(\tilde{p}_1, \ldots, \tilde{p}_n)$ that include all intermediate-degree terms that emerge in the circuit. During the second step, CP_0 calculates $r_1^{\tilde{p}_1} \cdots r_n^{\tilde{p}_n}$ for each $(\tilde{p}_1, \ldots, \tilde{p}_n) \in \mathcal{P}$ as well as $f_j(r_1, \ldots, r_n)$ for each $j \in \{1, \ldots, \ell\}$ and secret-shares these results with CP_1 and CP_2 in the final step. This provides all necessary terms for CP_1 and CP_2 to non-interactively compute $[f_j(x_1, \ldots, x_n)]$ for every $j \in \{1, \ldots, \ell\}$ using Eq. (3).

The linearization cost. We define the *linearization cost* $\tilde{T}$ of an arithmetic circuit to be the cardinality of the aforementioned union set $\mathcal{P}$ of intermediate-degree terms. Intuitively, $\tilde{T}$ represents the number of extra terms that need to be calculated by CP_0 (over the blinding factors) and secret-shared with CP_1 and CP_2 as a consequence of linearizing the function as in Eq. (3). This notion will

be useful in analyzing the efficiency of our Beaver partitioning approach when applied to various arithmetic circuits.

3.1 Comparison with Beaver Multiplication Triples

In Supplementary Table 4, we compare the communication complexity of our method with the standard Beaver multiplication triple method that invokes Protocol 7 for each multiplication gate in the arithmetic circuit. For both approaches, we allow CP_0 to precompute all required operations and transfer the shares to CP_1 and CP_2 in advance, since its computation is input-independent. Also, we allow data transfer to be performed in batches to minimize communication rounds. For instance, this allows the baseline approach to achieve communication rounds equal to the multiplicative depth of the circuit, by batching Lines 2 and 3 of Protocol 7 across different multiplication gates that lie in the same "layer" of the circuit (mutually independent given the output of previous layers).

	Beaver multiplication triples	**Our method**
Rounds (online)	d	1
Bandwidth (online), $\mathsf{CP}_1 \leftrightarrow \mathsf{CP}_2$	$2m$	k
Bandwidth (offline), $\mathsf{CP}_0 \rightarrow \mathsf{CP}_{1/2}$	$3m$	$k + \ell + \tilde{T}$

Supplementary Table 4: Comparison of Beaver multiplication triples and our generalized method for securely evaluating an arithmetic circuit with k inputs, ℓ outputs, multiplicative depth d, m multiplication gates, and linearization cost $\tilde{T}$ (defined in the previous section).

The key advantage of our generalized method is that irrespective of the number of multiplication gates and depth of the circuit, the online phase only requires a single round of communication and bandwidth equal to the input size. As a tradeoff, however, we incur a potentially large offline communication cost that includes $\tilde{T}$, which is $O(2^d)$ in general. We address this tradeoff by employing our method only in situations where the benefits clearly outweigh the costs. In the following, we highlight three concrete scenarios where our generalized Beaver partitioning approach is useful in the context of performing large-scale GWAS.

3.2 Scenario 1: Depth-One Circuits (e.g., Matrix Multiplication)

When the multiplicative depth of the circuit is at most one ($\deg(f) \leq 1$), there are no intermediate-degree terms in Eq. (1), and the linearization cost becomes zero ($\tilde{T} = 0$). A frequent operation in GWAS for which this property holds is matrix multiplication. Suppose we want to compute the product of two secret-shared matrices with dimensions $d_1 \times d_2$ and $d_2 \times d_3$. Even though this operation requires evaluating $m = d_1 d_2 d_3$ pairwise multiplications, the multiplicative depth d of the overall computation is 1. Moreover, the arithmetic circuit for performing matrix multiplication contains $d_1 d_2 + d_2 d_3$ inputs and $d_1 d_3$ outputs. Substituting these values into the expression in Supplementary Table 4, we obtain complexities as shown in Supplementary Table 5. Notably, using our Beaver partitioning technique, both the online and offline bandwidth is proportional to the size of the matrices (i.e., quadratic in the dimensions) rather than the number of multiplications (i.e., cubic in the dimensions). This difference is critical for building a practical GWAS protocol that can support millions of individuals and SNPs.

	Multiplication triples	Our method
Rounds (online)	1	1
Bandwidth (online), $\mathsf{CP}_1 \leftrightarrow \mathsf{CP}_2$	$2d_1d_2d_3$	$d_1d_2 + d_2d_3$
Bandwidth (offline), $\mathsf{CP}_0 \rightarrow \mathsf{CP}_{1/2}$	$3d_1d_2d_3$	$d_1d_2 + d_2d_3 + d_1d_3$

Supplementary Table 5: Comparison of Beaver multiplication triples and our generalized method for securely multiplying $d_1 \times d_2$ and $d_2 \times d_3$ matrices.

3.3 Scenario 2: Variable Reuse

Because our generalized Beaver partitioning method can evaluate circuits with multiple outputs, multiple operations (each represented as a circuit) that share common inputs can be combined into a single circuit for joint evaluation. In doing so, each input only needs to be Beaver partitioned once, which is more efficient than constructing a fresh Beaver partition for each operation individually. In fact, we can implicitly combine circuits operating on common inputs by reusing the Beaver partitions on the common inputs across multiple invocations of Protocol 9.

More precisely, given a secret-shared input $[x]$ involved in the evaluation of $f_1(x, \dots)$ followed by $f_2(x, \dots)$, we can simulate the joint evaluation of f_1 and f_2 by obtaining the Beaver partition of $[x]$ during the evaluation of f_1 and reusing the results, namely $\langle x - r, x - r \rangle$ and $\langle [r]_1, [r]_2, r \rangle$ for a uniformly and independently random $r \in \mathbb{Z}_q$, for f_2. Note this does not affect the security of our method, because in every invocation of $\mathsf{BeaverPartition}$ or $\mathsf{ShareSecret}$ in the overall protocol, the views of CP_1 and CP_2 still consist only of uniform and independent values in $\mathbb{Z}_q$.

As a concrete example, consider the following power iteration procedure that appears in our randomized PCA protocol (Protocol 32):

for each t in $1, \dots, T$ **do**
  $A^{(t)} \leftarrow G^T G A^{(t-1)}$
end for

where G is $d_1 \times d_2$ and $A^{(t)}$ is $d_2 \times d_3$ for all t. In practice, d_1 and d_2 are large while d_3 is small. Even with our improved method for secure matrix multiplication (Scenario 1), carrying out this computation by considering each multiplication separately would require Beaver partitioning G a total of $2T$ times—a prohibitive cost for a large G. With the reuse of Beaver partitioned data, we need to Beaver partition G only once. This is another key factor in achieving practicality for large-scale GWAS. The complexity comparison for this example is shown in Supplementary Table 6.

	Beaver triples	Our method (without reuse)	Our method (with reuse)
Rounds (online)	$2T$	$2T$	$2T$
Bandwidth (online), $\mathsf{CP}_1 \leftrightarrow \mathsf{CP}_2$	$4d_3Td_1d_2$	$2Td_1d_2 + d_3T(d_1 + d_2)$	$d_1d_2 + d_3T(d_1 + d_2)$
Bandwidth (offline), $\mathsf{CP}_0 \rightarrow \mathsf{CP}_{1/2}$	$6d_3Td_1d_2$	$2Td_1d_2 + 2d_3T(d_1 + d_2)$	$d_1d_2 + 2d_3T(d_1 + d_2)$

Supplementary Table 6: Comparison of Beaver multiplication triples and our generalized method for power iteration (shown in text) with an $d_1 \times d_2$ matrix G and $d_2 \times d_3$ matrices $A^{(t)}$ for all t with $d_3 \ll d_1, d_2$. Both our methods are based on invoking our improved matrix multiplication (see Scenario 1) $2T$ times.

3.4 Scenario 3: Exponentiation

Exponentiation is a special case where the linearization cost grows linearly with the circuit depth. Specifically, consider a circuit that takes a single value x as input and outputs all powers of x up to a known parameter $\alpha \geq 2$: $x^2, \ldots, x^\alpha$. For simplicity, assume that the circuit is naïvely represented as a sequence of $\alpha - 1$ pairwise multiplications. In this case, the circuit has 1 input, $\alpha - 1$ outputs, depth $\alpha - 1$, and consists of $\alpha - 1$ multiplication gates. Its linearization cost $\tilde{T}$ is $\tilde{T} = \alpha - 2$, since there are $\alpha - 2$ intermediate powers of the blinding factor (from 2 to $\alpha - 1$) that needs to be calculated and shared by CP_0. Using the asymptotic analysis given in Supplementary Table 4, we compare our Beaver partitioning method to the standard method of directly applying multiplication triples in Supplementary Table 7. Our method has the advantage that it is single-round (compared to $\alpha - 1$ rounds), and the online communication consists of a single field element (compared to $2(\alpha - 1)$). The offline bandwidth is still linear in the exponent α, though the constants are modestly smaller. The online savings in communication and round complexity leads to a significant increase in efficiency in our GWAS protocol, as this subroutine (Protocol 10) is frequently invoked as part of the secure table lookup operation (Protocol 11).

	Multiplication triples	**Our method**
Rounds (online)	$\alpha - 1$	1
Bandwidth (online), $\mathsf{CP}_1 \leftrightarrow \mathsf{CP}_2$	$2(\alpha - 1)$	1
Bandwidth (offline), $\mathsf{CP}_0 \rightarrow \mathsf{CP}_{1/2}$	$3(\alpha - 1)$	$2(\alpha - 1)$

Supplementary Table 7: Comparison of Beaver multiplication triples and our generalized method for securely evaluating an exponentiation function that outputs all powers of input up to a given exponent α.

Supplementary Note 4: Our Protocol Building Blocks

Here, we introduce a number of protocol building blocks that we leverage when constructing our GWAS protocol.

4.1 Table Lookup

Let $\mathsf{T} = \{k_i \mapsto v_i\}_{i=1}^{d}$ be a public table that maps keys k_i to values v_i, where $k_i, v_i \in \mathbb{Z}_q$ for all $i \in \{1, \ldots, d\}$. The table lookup functionality takes as input a key $k \in \mathbb{Z}_q$ and returns a value $v \in \mathbb{Z}_q$, where $v = \mathsf{T}(k)$ if k is in the table. If k is not present in the table, then the output of the table lookup is undefined (can be an arbitrary field element). A table lookup on secret-shared inputs corresponds to the setting where the input key $[k]$ is secret-shared, and the output should be a secret sharing of the value $[v]$. Table lookups can be implemented by representing the entries in the table as points on a polynomial f. Then, looking up a key k in T just corresponds to evaluating the polynomial on k. We write $\mathsf{LagrangeInterpolation}$ to denote an algorithm that takes as input a table T of d values and outputs a polynomial (of degree $d-1$) that interpolates the mappings in T:

- $\mathsf{LagrangeInterpolation}(\mathsf{T}) \to c_0, \ldots, c_{d-1}$: On input a table $\mathsf{T} = \{k_i \mapsto v_i\}_{i=1}^{d}$, the Lagrange interpolation algorithm return the coefficients $c_0, \ldots, c_{d-1}$ of a polynomial $f(x) = \sum_{i=0}^{d-1} c_i x^i$ where $f(k_i) = v_i$ for all $i \in \{1, \ldots, d\}$.

By representing tables as polynomials, table lookup just corresponds to polynomial evaluation. To perform polynomial evaluation on a secret-shared input, we first define a subprotocol Powers that securely computes all powers of input up to a given exponent (Protocol 10) using our generalized Beaver partitioning approach (Supplementary Note 3). After obtaining $[k^t]$ for $t = 2, \ldots, d-1$ via this subroutine, the interpolated polynomial can be easily evaluated as an affine function $\sum_{t=0}^{d-1} c_t[k^t]$. This procedure is summarized in Protocol 11.

Protocol 10: $[x^2], \ldots, [x^d] \leftarrow \mathsf{Powers}([x], d)$

Require: $d \geq 2$

1: $\langle x - r, x - r \rangle, \langle [r]_1, [r]_2, r \rangle \leftarrow \mathsf{BeaverPartition}([x])$
2: CP_0: Calculate $r^2, \ldots, r^d \in \mathbb{Z}_q$
3: **for each** t in $\{2, \ldots, d\}$ **do** (in parallel)
4: $\quad [r^t] \leftarrow \mathsf{ShareSecret}(\langle \perp, \perp, r^t \rangle)$
5: **end for**
6: **for each** t in $\{2, \ldots, d\}$ **do**
7: $\quad [x^t] \leftarrow (x-r)^t + \sum_{i=1}^{t} \binom{t}{i} (x-r)^{t-i} [r^i]$
8: **end for**
9: **return** $[x^2], \ldots, [x^d]$

Protocol 11: $[\mathsf{T}(k)] \leftarrow \mathsf{TableLookup}(\mathsf{T} = \{k_i \mapsto v_i\}_{i=1}^{d}, [k])$

Require: $k \in \{k_1, \ldots, k_d\}, d \geq 2$

1: $c_0, \ldots, c_{d-1} \leftarrow \mathsf{LagrangeInterpolation}(\mathsf{T})$
2: $[k^2], \ldots, [k^{d-1}] \leftarrow \mathsf{Powers}([k], d-1)$
3: **return** $c_0 + c_1[k] + \cdots + c_{d-1}[k^{d-1}]$

Nature Biotechnology: doi:10.1038/nbt.4108

4.2 Bitwise Operations

We now describe several protocols for performing bit-wise operations on secret-shared inputs. Let $\{[a_i]\}_{i=1}^{L} := \{[a_1], ..., [a_L]\}$ be a secret L-bit vector where $a_i \in \{0, 1\}$ for all i, and every individual bit is secretly shared.

- The FanInOr subroutine (Protocol 12) securely computes the disjunction ("or" operation) of all of the bits by performing a table lookup using the sum of bits as the key (i.e., a table where the value 0 maps to 0 and all other values map to 1).
- Next, the PrefixOr subroutine (Protocol 13) efficiently computes FanInOr of all L prefixes of the bit vector in a way that does not require invoking FanInOr on each prefix [10]. The communication in the optimized protocol scales linearly in L (instead of quadratically in L in the case of the naïve protocol). Note that Lines 12-14 of Protocol 13 can be carried out as a group using our Beaver partitioning method to increase efficiency, since it is a depth-one circuit (Supplementary Note 3, Section 3.2). The same optimization applies to Lines 18-20.
- BinaryLessThan (Protocol 14) is a subroutine that takes the binary representation of two integers (i.e., bit vectors) and returns a share of the comparison result. Our protocol is based on Damgård et. al [11]. The main technique used is drawn from a well-known solution of Yao's millionaire problem [12]. Note that the Lines 1-3 and 5 of Protocol 14 are depth-1 arithmetic computations, and thus, can be more efficiently implemented using our Beaver partitioning method. Moreover, when one of the numbers being compared is known to both CP_1 and CP_2, both Lines 1-3 and 5 turn into affine functions over the secret inputs, and thus can be calculated non-interactively. This special case is described in BinaryLessThanPublic (Protocol 15).

Protocol 12: $[\vee_{i=1}^{n} a_i] \leftarrow \mathsf{FanInOr}(\{[a_i]\}_{i=1}^{n})$ (based on [11])

Require: $a_i \in \{0, 1\}, \forall i$

1: $[s] \leftarrow 1 + \sum_{i=1}^{n} [a_i]$
2: $[v] \leftarrow \mathsf{TableLookup}(\{1 \rightarrow 0\} \cup \{i \rightarrow 1\}_{i=2}^{n+1}, [s])$
3: **return** $[v]$

Protocol 13: $\{[\vee_{j=1}^{i} a_j]\}_{i=1}^{L} \leftarrow \mathsf{PrefixOr}(\{[a_i]\}_{i=1}^{L})$ (based on [11])

Require: $a_i \in \{0,1\}, \forall i$ and assume $L = \ell^2$ for an integer ℓ (if not, pad with leading zeros)

Reshape input as a ℓ-by-ℓ matrix $[A]$ where $[A_{i,j}] = [a_{(i-1)\ell+j}]$
for each i in $\{1, \ldots, \ell\}$ **do** (in parallel)
    $[x_i] \leftarrow \mathsf{FanInOr}(\{[A_{i,t}]\}_{t=1}^{\ell})$
end for
for each i in $\{1, \ldots, \ell\}$ **do** (in parallel)
    $[y_i] \leftarrow \mathsf{FanInOr}(\{[x_t]\}_{t=1}^{i})$
end for
$[f_1] \leftarrow [x_1]$
for each i in $\{2, \ldots, \ell\}$ **do**
    $[f_i] \leftarrow [y_i] - [y_{i-1}]$
end for
for each j in $\{1, \ldots, \ell\}$ **do** (in parallel)
    $[c_j] \leftarrow \sum_{i=1}^{\ell} [f_i][A_{i,j}]$
end for
for each j in $\{1, \ldots, \ell\}$ **do** (in parallel)
    $[d_j] \leftarrow \mathsf{FanInOr}(\{[c_t]\}_{t=1}^{j})$
end for
for each (i,j) in $\{1, \ldots, \ell\}^2$ **do** (in parallel)
    $[b_{(i-1)\ell+j}] \leftarrow [f_i][d_j] + [y_i] - [f_i]$
end for
return $\{[b_i]\}_{i=1}^{L}$

Protocol 14: $[a <_? b] \leftarrow \mathsf{BinaryLessThan}(\{[a_i]\}_{i=1}^{L}, \{[b_i]\}_{i=1}^{L})$ (based on [11])

Require: $a_i, b_i \in \{0,1\}, \forall i$, and index starts from the most significant bit

for each i in $\{1, \ldots, L\}$ **do** (in parallel)
    $[a_i \oplus b_i] \leftarrow [a_i] - 2[a_i][b_i] + [b_i]$
end for
$\{[f_i]\}_{i=1}^{L} \leftarrow \mathsf{PrefixOr}(\{[a_i \oplus b_i]\}_{i=1}^{L})$
$[c] \leftarrow [b_1][f_1] + \sum_{i=2}^{L} [b_i]([f_i] - [f_{i-1}])$
return $[c]$

Protocol 15: $[a <_? b] \leftarrow \mathsf{BinaryLessThanPublic}(\{[a_i]\}_{i=1}^{L}, \{\langle b_i, b_i \rangle\}_{i=1}^{L})$

Require: $a_i, b_i \in \{0,1\}, \forall i$, and index starts from the most significant bit

for each i in $\{1, \ldots, L\}$ **do**
    $[a_i \oplus b_i] \leftarrow [a_i] - 2b_i[a_i] + b_i$
end for
$\{[f_i]\}_{i=1}^{L} \leftarrow \mathsf{PrefixOr}(\{[a_i \oplus b_i]\}_{i=1}^{L})$
$[c] \leftarrow b_1[f_1] + \sum_{i=2}^{L} b_i([f_i] - [f_{i-1}])$
return $[c]$

Nature Biotechnology: doi:10.1038/nbt.4108

Supplementary Note 5: Computing with Fixed-Point Numbers

To use secret sharing in practice, we need to define a mapping between data values and field elements. In this work, we use a fixed-point (FP) representation [13] to represent real-valued numbers that appear during the GWAS computation. Various building blocks for secure computation with FP numbers, including division, have been proposed by Catrina and Saxena [13]. Here, we recall their protocols and describe several optimizations that we make in our work (namely, using a server-aided design and removing the dependence on the secure bit-decomposition subroutine). We additionally describe novel square-root and square-root inversion subroutines, which are frequently invoked in GWAS.

5.1 Data Representation

Let k be the number of bits we use to represent a signed real number, of which f bits are allocated to the fractional domain (referred to as the *precision*). The values of k and f are chosen such that any real number we expect to encounter during the protocol falls in the range

$$\mathcal{D} := [-2^{k-f-1}, 2^{k-f-1}) \subset \mathbb{R}.$$

We map each $x \in \mathcal{D}$ to an element in the finite field $\mathbb{Z}_q$ using the encoding function

$$E_f(x) = \lfloor x \cdot 2^f \rfloor \bmod q,$$

where $\lfloor \cdot \rfloor$ denotes the floor function. Conversely, each field element $z \in \mathbb{Z}_q$ is mapped back to real data space $\mathcal{D} \cup \{\mathsf{NaN}\}$ using the following decoding function:

$$D_f(z) = \begin{cases} z \cdot 2^{-f} & \text{if } 0 \le z < 2^{k-1}, \\ -(q-z) \cdot 2^{-f} & \text{if } q - 2^{k-1} \le z \le q-1, \\ \mathsf{NaN} & \text{otherwise.} \end{cases}$$

Intuitively, this representation corresponds to taking a real number in $\mathcal{D}$ and truncating it to the closest multiple of 2^{-f}.

Notation. We write $[E_f(x)]$ to denote a secret-share of a value $x \in \mathcal{D}$. We alternatively express this as

$$[x]^{(f)} := [E_f(x)] \text{ for } x \in \mathcal{D}.$$

We use the same base field $\mathbb{Z}_q$ for all values of f for interoperability. We choose q to be large enough for the highest precision required. Secret sharing of integers $x \in \mathcal{D}$ with an identity encoding function (i.e., $[x]^{(0)}$) is denoted without the superscript as $[x]$, i.e.,

$$[x] := [x]^{(0)} \text{ for } x \in \mathcal{D} \cap \mathbb{Z},$$

since it is equivalent to directly treating the integer as an element in $\mathbb{Z}_q$.

5.2 Arithmetic Operations

Addition and subtraction of secret-shared fixed-point values correspond to addition and subtraction on the underlying secret-shared field elements, provided that the fixed-point values are encoded with the same precision. In particular, we define

$$[x \pm y]^{(f)} := [x]^{(f)} \pm [y]^{(f)} \text{ and } [x \pm a]^{(f)} := [x]^{(f)} \pm E_f(a),$$

for secret-shared values $x, y \in \mathcal{D}$ and public constant $a \in \mathcal{D}$. Unlike the case of adding and subtracting shares of field elements, adding and subtracting shares of encoded fixed point values introduces a small amount of error (bounded by $1/2^{f-1}$). To see why, observe that by construction, the shared value $[x]^{(f)} + [y]^{(f)}$ is an additive sharing of $D_f(E_f(x)) + D_f(E_f(y))$. Due to the imprecision introduced by the fixed-point encoding, this value is close to $[x+y]^{(f)} = D_f(E_f(x+y))$, but there is some error due to the non-linearity of the encoding/decoding procedures.

Multiplying two FP numbers requires an additional step. Note that directly multiplying two encoded FP numbers outputs a result with precision $2f$ instead of f. In particular, we write

$$[xy]^{(2f)} := [x]^{(f)}[y]^{(f)}.$$

To scale the precision back to f, we use the Truncate subroutine (Protocol 16) from Catrina and Saxena [13]. Given public values b and s, this protocol truncates the s least significant bits of a secret-shared integer (in our case, the encoding of a FP number). The output is a secret-sharing of the most significant b bits of the truncated input. The resulting secure multiplication protocol for FP numbers, MultiplyFP, is shown in Protocol 17.

Protocol 16: $[x_{\text{trunc}}] \leftarrow \mathsf{Truncate}([x], b, s)$

Require: Number of bits to mask b, number of least significant bits to truncate s, and a statistical security parameter κ

1: CP_0: Sample $r \xleftarrow{R} \{0, \ldots, 2^{b+\kappa} - 1\}$
2: CP_0: Calculate $r_{\text{tail}} \leftarrow r \mod 2^s$
3: $[r] \leftarrow \mathsf{ShareSecret}(\langle \perp, \perp, r \rangle)$ (in parallel with Line 4)
4: $[r_{\text{tail}}] \leftarrow \mathsf{ShareSecret}(\langle \perp, \perp, r_{\text{tail}} \rangle)$
5: $\langle x + r, x + r \rangle \leftarrow \mathsf{ReconstructSecret}([x] + [r])$
6: $\mathsf{CP}_1, \mathsf{CP}_2$: Calculate $(x + r)_{\text{tail}} \leftarrow x + r \mod 2^s$
7: **return** $(2^s)^{-1} \cdot ([x] + [r_{\text{tail}}] - (x + r)_{\text{tail}})$, where $(2^s)^{-1}$ denotes the inverse of 2^s in $\mathbb{Z}_q$

Security. Notably, Line 5 of Truncate reveals the value $x+r$ in the clear (to CP_1 and CP_2), where r is uniform over the set $\{0, \ldots, 2^{b+\kappa} - 1\}$, and $x \in \{0, \ldots, 2^b - 1\}$. Assume that the base prime q is large enough such that $x+r$ does not overflow. Then, the distribution of $x+r$ is statistically close to the uniform distribution over $\{0, \ldots, 2^{b+\kappa} - 1\}$. More precisely, the statistical distance[1] between the two distributions is bounded by $2^{-\kappa}$ for all $x \in \{0, \ldots, 2^b - 1\}$. This implies no adversary (including computationally unbounded ones) is able to distinguish $x + r$ from a uniformly random field element r', except with an *advantage* $2^{-\kappa}$—i.e., the difference in the *a posteriori* guessing probability of a given candidate being a real message (i.e., $x + r$) between the two candidates (i.e., r' and $x + r$, randomly shuffled) is upper-bounded by $2^{-\kappa}$, where 0 corresponds to perfect indistinguishability. Thus, CP_1 and CP_2 effectively sees $x + r$ as a uniformly random element, and do not learn anything about x from it as a result.

Our overall GWAS protocol requires multiple invocations of the Truncate protocol, which increases the adversary's distinguishing advantage by a multiplicative factor v equal to the total number of invocations (based on a standard hybrid argument; cf. [14, §3.2.3]). In practice, we choose our statistical security parameter κ such that the adversary's distinguishing advantage $v \cdot 2^{-\kappa} < 2^{-30}$, or equivalently, $\kappa - \log_2(v) > 30$. This ensures that the view of each party in the *overall* protocol is statistically indistinguishable from a sequence of uniformly random field elements.

[1] The statistical distance between two distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ over a common finite domain $\mathcal{X}$ is defined to be $\frac{1}{2} \sum_{x \in \mathcal{X}} |\Pr[\mathcal{D}_1(x)] - \Pr[\mathcal{D}_2(x)]|$.

Next, since $2^{b+\kappa}$ lower-bounds the base prime q we use for the overall protocol, which in turn determines the overall efficiency, we choose the smallest possible b for each invocation of Truncate. For example, in Line 2 of MultiplyFP, we choose $b = k + f$. Recall that by assumption, every value encountered during the protocol lies in $\mathcal{D}$. This means that $xy \in \mathcal{D}$, which implies $E_{2f}(xy)$ has effective bit-length at most $k + f$.

Protocol 17: $[xy]^{(f)} \leftarrow \mathsf{MultiplyFP}([x]^{(f)}, [y]^{(f)})$

1: $[z]^{(2f)} \leftarrow [x]^{(f)}[y]^{(f)}$
2: $[w]^{(f)} \leftarrow \mathsf{Truncate}([z]^{(2f)}, k + f, f)$
3: **return** $[w]^{(f)}$

5.3 Field Conversion

The following sections describe protocols in which individual bits of a secret integer are separately secret-shared. To exploit the smaller field size needed for bit operations compared to FP operations, we instead perform operations over an auxiliary field $\mathbb{Z}_{q'}$ where $q' \ll q$. We write $[[x]] := \langle [[x]]_1, [[x]]_2 \rangle$ to denote a secret sharing over $\mathbb{Z}_{q'}$. Because our use of the auxiliary field is limited to unsigned integers, the mapping between the data space and the auxiliary field is simply the identity map over $\{0, \ldots, q' - 1\}$. When appropriate, we invoke subroutines previously defined over $[\cdot]$ with the secret shares $[[\cdot]]$. In our protocols, secret shares over $\mathbb{Z}_{q'}$ are constructed using the function ShareSecretSmallField defined as follows:

- $\mathsf{ShareSecretSmallField}(\langle \bot, \bot, x \rangle, q') \rightarrow [[x]]$: Same as ShareSecret (Protocol 1) except x is shared in the smaller field $\mathbb{Z}_{q'}$.

After operating on the secret-shared bits, the resulting secret shares can be mapped back into the large field $\mathbb{Z}_q$ via TableLookupWithFieldConversion (Protocol 18). In other words, the TableLookupWithFieldConversion algorithm converts a secret sharing of $x \in \mathbb{Z}_{q'}$ into a secret-sharing of $x \in \mathbb{Z}_q$. The field conversion algorithm relies on the observation that if we simply view $[[x]]$ as a secret sharing of x over the large field $\mathbb{Z}_q$, then performing the reconstruction (namely, computing $[[x]]_1 + [[x]]_2$ in $\mathbb{Z}_q$) either yields the value $x \in \mathbb{Z}_q$ or the value $x + q' \in \mathbb{Z}_q$. We can thus use TableLookup to map each of these two possibilities for x' to the value $x \in \mathbb{Z}_q$. Of course, this approach is efficient only if there are just a few possible values to convert (e.g., if $\mathbb{Z}_{q'}$ is small). This is indeed the setting of our protocol, since the outputs of bit operations lie in a small range.

Protocol 18: $[v] \leftarrow \mathsf{TableLookupWithFieldConversion}(\mathsf{T} = \{k_i \mapsto v_i\}_{i=1}^{d}, [[k]])$

Require: $k \in \{k_1, \ldots, k_d\}$, $d \geq 2$, $[[x]]$ secret-shared in $\mathbb{Z}_{q'}$ with $q' \ll q$
1: $\mathsf{T}' \leftarrow \mathsf{T} \cup \{(k_i + q') \mapsto v_i \colon (k_i \mapsto v_i) \in \mathsf{T}\}$
2: $[k'] \leftarrow \langle [[k]]_1, [[k]]_2 \rangle$
3: $[v] \leftarrow \mathsf{TableLookup}(\mathsf{T}', [k'])$
4: **return** $[v]$

5.4 Comparison

To implement a comparison protocol on secret-shared fixed-point values, we first implement a secure sign test for FP values (i.e., a comparison with zero). Here, we use the technique of Nishide and

Ohta [15] for comparing elements of $\mathbb{Z}_q$ to the value $q/2$ (as a real number). Given a secret-shared FP value $[x]^{(f)}$, the idea is to securely retrieve the least significant bit of $2 \cdot E_f(x)$. If x is negative, then $E_f(x) > q/2$, and thus $2 \cdot E_f(x)$ wraps around and results in an odd integer (since $q > 2$ is odd). On the other hand, if x is positive, then $E_f(x) < q/2$, which implies $2 \cdot E_f(x)$ is even. Thus, retrieving the least significant bit of $2 \cdot E_f(x)$ is equivalent to performing a sign test on x. The least significant bit is securely obtained by manipulating the binary representation of $2 \cdot E_f(x) + r$, where the blinding factor r is sampled uniformly at random and secret-shared by CP_0. The overall procedure is shown in IsPositive (Protocol 19). Note we make use of the following helper functions:

- $\mathsf{BitLength}(x) \to \ell$: Returns the bit length of an integer x.
- $\mathsf{BinaryRepresentation}(x, L) \to \{a_i\}_{i=1}^{L}$: Takes a field element $x \in \mathbb{Z}_q$ and returns the L least significant bits $a_1, \ldots, a_L$ of its binary representation in decreasing order of significance.

Protocol 19: $[x >_? 0] \leftarrow \mathsf{IsPositive}([x]^{(f)})$ (based on [15])

Require: Base prime of main field q, base prime of auxiliary field q_2

1: $L \leftarrow \mathsf{BitLength}(q)$
2: CP_0: Sample $r \xleftarrow{\text{R}} \mathbb{Z}_q$
3: CP_0: Calculate $\{a_i\}_{i=1}^{L} \leftarrow \mathsf{BinaryRepresentation}(r, L)$
4: $[r] \leftarrow \mathsf{ShareSecret}(\langle \perp, \perp, r \rangle)$
5: **for each** i in $\{1, \ldots, L\}$ **do** (in parallel, including Line 4)
6: $\quad [[a_i]] \leftarrow \mathsf{ShareSecretSmallField}(\langle \perp, \perp, a_i \rangle, q_2)$
7: **end for**
8: $\langle 2 \cdot E_f(x) + r, 2 \cdot E_f(x) + r \rangle \leftarrow \mathsf{ReconstructSecret}(2[x]^{(f)} + [r])$
9: $\mathsf{CP}_{1,2}$: Calculate $\{b_i\}_{i=1}^{L} \leftarrow \mathsf{BinaryRepresentation}(2E_f(x) + r, L)$
10: $[[a <_? b]] \leftarrow \mathsf{BinaryLessThanPublic}(\{[[a_i]]\}_{i=1}^{L}, \{\langle b_i, b_i \rangle\}_{i=1}^{L})$
11: $[[a_L \oplus b_L]] \leftarrow [[a_L]] - 2b_L[[a_L]] + b_L$
12: $[[u]] \leftarrow [[a_L \oplus b_L]] \cdot [[a <_? b]]$
13: $[s] \leftarrow \mathsf{TableLookupWithFieldConversion}(\{1 \to 1, 2 \to 0\}, [[u]] + 1)$
14: **return** $[s]$

Security. The security of IsPositive follows from the security of secret sharing (Lines 4-7) and the fact that $2E_f(x) + r$ in Line 8 does not reveal any information about x to either party, since r is uniformly random and independent from all other values.

Comparing secret-shared FP values. The IsPositive protocol can be used to compare two secret-shared FP values $[x]^{(f)}$ and $[y]^{(f)}$ by casting the problem as a sign test of the difference $[y]^{(f)} - [x]^{(f)}$. This extended protocol is named LessThan (Protocol 20). Additionally, we define LessThanPublic (Protocol 21) for the setting where one of the input values is known to both CP_1 and CP_2.

Protocol 20: $[x <_? y] \leftarrow \mathsf{LessThan}([x]^{(f)}, [y]^{(f)})$

Require: x and y have the same sign

1: $[c] \leftarrow \mathsf{IsPositive}([x]^{(f)} - [y]^{(f)})$
2: **return** $1 - [c]$

Protocol 21: $[x <_? y] \leftarrow \mathsf{LessThanPublic}([x]^{(f)}, \langle y, y \rangle)$

Require: x and y have the same sign, $y \in \mathcal{D}$

1: $[c] \leftarrow \mathsf{IsPositive}([x]^{(f)} - E_f(y))$
2: **return** $1 - [c]$

5.5 Division and Square Root

We implement division and square root routines (for FP values) using Goldschmidt's algorithm [16], which approximates the desired operation as a series of multiplications. Each iteration quadratically reduces the relative approximation error.

Goldschmidt's algorithm for division. Take two real numbers $a, b \in \mathbb{R}$. We describe Goldschmidt's algorithm for approximating the quotient $a/b \in \mathbb{R}$. Let w_0 be an initial approximation of $1/b$ and let $\epsilon_0 := 1 - bw_0$ be the relative error for the approximation w_0. We require that $|\epsilon_0| < 1$. The algorithm iteratively computes the following:

$$x_t \leftarrow x_{t-1}(1 + y_{t-1})$$
$$y_t \leftarrow y_{t-1}y_{t-1}$$

with $x_0 = aw_0$ and $y_0 = \epsilon_0$. Here, x_t converges to a/b [16].

Goldschmidt's algorithm for square root. For computing the square root of a value $a \in \mathbb{R}$, Goldschmidt's algorithm starts with an initial approximation w_0 of $1/\sqrt{a}$. As before, let $\epsilon_0 := 1 - aw_0^2$ denote the relative error in the approximation. We require that $|\epsilon_0| < 1$. The algorithm then iteratively computes

$$z_{t-1} \leftarrow 1.5 - x_{t-1}y_{t-1}$$
$$x_t \leftarrow z_{t-1}x_{t-1}$$
$$y_t \leftarrow z_{t-1}y_{t-1}$$

with $x_0 = aw_0$ and $y_0 = w_0/2$. Here, x_t converges to $\sqrt{a}$ and $2y_t$ converges to $1/\sqrt{a}$ [16]. Note we obtain the square root inverse of a for free.

Choosing the number of iterations. Normally, if the input is available in the clear, we would iterate Goldschmidt's algorithm until the values satisfy some convergence criterion. However, this strategy does not extend to the secure computation setting when the number of iterations needed before convergence could itself reveal information about the underlying inputs. In our protocol execution, we thus take an *oblivious* evaluation approach and instead fix the number of iterations in advance and always iterate the algorithm for that many rounds. We choose the number of iterations to be sufficiently large to ensure the desired level of accuracy. In this work, we follow the recommendation from a previous work [13] and use $2\lceil \log_2(k/3.5) \rceil$ iterations for both the division and square root routines. An extra iteration is added for division protocol to account for our initial approximation (described below) being slightly less accurate, albeit more efficient, than what was proposed in the original work.

Computing the initial approximations. For both the division and square root routines, we require a suitable initial approximation w_0 of $1/x$ or $1/\sqrt{x}$ to ensure convergence of the iterative procedure. We achieve this by first scaling the input to be within $[0.25, 1)$ and evaluating a quadratic polynomial that approximates the desired function within that range. Reverse scaling is performed to map the approximation back to the scale of input.

We introduce a new protocol NormalizerEvenExponent (Protocol 22) to efficiently handle the scaling operation, which is based on the constant-round bit length protocol of Dahl et. al [17]. Notably, our protocol replaces the expensive bit decomposition routine that previous work relied on [13]. The protocol NormalizerEvenExponent takes as input a secret-shared integer $[z]$, and returns two secret numbers $[2^{2t}]$ and $[2^t]$ such that $2^{k-2} \leq 2^{2t} z < 2^k$. Since the algorithm is rather involved, we refer the readers to the [17, Appendix A.1] for a description of the technique. In this work, we modify their algorithm to return the normalization factor instead of the bit-length of the input and added an extra step of finding the closest even power of two, which is achieved by splitting a bit vector into even and odd bits (Lines 22 and 23 of NormalizerEvenExponent). Similar to Truncate, the NormalizerEvenExponent protocol achieves statistical security (with statistical security parameter κ) as opposed to perfect security. In particular, Line 7 of NormalizerEvenExponent blinds the input value x (which has at most k bits) with a uniformly random value from $\{0, \ldots, 2^{k+\kappa} - 1\}$. The distribution of the blinded value $x + r$ is at least 2^{κ}-close to the uniform distribution over $\{0, \ldots, 2^{k+\kappa} - 1\}$.

After obtaining $[2^{2t}]$ and $[2^t]$ where $2^{k-2} \leq 2^{2t} \cdot z < 2^k$ for the input value $[x]^{(f)}$, we scale x to $[0.25, 1)$ by truncating $k - f$ bits of $[2^{2t}][x]^{(f)}$ with Truncate. The resulting scaled input is denoted $[\bar{x}]^{(f)}$. We only need to mask k bits during Truncate (i.e., $b = k$), since $2^{2t} \cdot E_f(x) < 2^k$. Note that we assume $x > 0$ during this process, which is always the case in our GWAS protocol. One can additionally support $x < 0$ by performing an additional sign test and providing $[|x|]^{(f)}$ as input to NormalizerEvenExponent.

Given the scaled input $[\bar{x}]^{(f)}$, we approximate the desired function by securely evaluating the quadratic polynomials

$$\frac{1}{\bar{x}} \approx 5.9430 - 10\bar{x} + 5\bar{x}^2 \quad \text{and} \quad \frac{1}{\sqrt{\bar{x}}} \approx 2.9581 - 4\bar{x} + 2\bar{x}^2,$$

where the coefficients of non-constant terms are chosen to be integers to avoid adding an extra call to Truncate. The resulting estimate, denoted $[\bar{w}_0]$, is scaled back to match the input as follows. For division, as noted in previous work [13], we can exploit the fact that

$$\frac{1}{x} = 2^{2t-(k-f)} \frac{1}{2^{2t-(k-f)} x} = 2^{2t-(k-f)} \frac{1}{\bar{x}} \approx 2^{2t-(k-f)} \bar{w}_0.$$

Thus, we approximate $1/x$ as

$$[w_0]^{(f)} \leftarrow \mathsf{Truncate}([2^{2t}][\bar{w}_0]^{(f)}, k + f + 2, k - f).$$

Note our scaled approximation $\bar{w}_0 < 4$ for $\bar{x} \in [0.25, 1)$. Thus, we set $b = k + f + 2$ for Truncate since $2^{2t} E_f(\bar{w}_0)$ has at most $k + f + 2$ bits.

Adapting the above approach for the inverse square root, we observe

$$\frac{1}{\sqrt{x}} = 2^{t - \frac{k-f}{2}} \frac{1}{\sqrt{2^{2t-(k-f)} x}} \approx 2^{t - \frac{k-f}{2}} \bar{w}_0,$$

which leads to

$$[w_0]^{(f)} \leftarrow \mathsf{Truncate}([2^t][\bar{w}_0]^{(f)}, \lfloor k/2 \rfloor + f + 2, (k - f)/2)$$

for approximating $1/\sqrt{x}$. Here, $b = \lfloor k/2 \rfloor + f + 2$ is sufficient, since 2^t is at most $\lfloor k/2 \rfloor$ bits and $\bar{w}_0 < 4$ as before. Importantly, we require $k - f$ to be even in order to ensure $(k-f)/2$ is an integer.

The resulting subroutines Divide and SqrtAndSqrtInverse are shown in Protocols 23 and 24. The security of both subroutines follow from the security of individual building blocks that are invoked during the procedure.

Protocol 22: $[2^{2t}], [2^t] \leftarrow \mathsf{NormalizerEvenExponent}([x])$ (adapts the bit-length protocol of [17])

Require: Data range $|x| < 2^{k-f-1}$ for even k, statistical security parameter κ, base prime of auxiliary field q_1

1: CP_0: Sample a random element $r \xleftarrow{\mathrm{R}} \{0, \ldots, 2^{k+\kappa} - 1\}$
2: CP_0: $\{r_i\}_{i=1}^k \leftarrow \mathsf{BinaryRepresentation}(r, k)$
3: $[r] \leftarrow \mathsf{ShareSecret}(\langle \bot, \bot, r \rangle)$
4: **for each** i in $\{1, \ldots, k\}$ **do** (in parallel, including Line 3)
5: $\quad [[r_i]] \leftarrow \mathsf{ShareSecretSmallField}(\langle \bot, \bot, r_i \rangle, q_1)$
6: **end for**
7: $\langle x + r, x + r \rangle \leftarrow \mathsf{ReconstructSecret}([x] + [r])$
8: $\mathsf{CP}_{1,2}$: $\{z_i\}_{i=1}^k \leftarrow \mathsf{BinaryRepresentation}(x + r, k)$
9: $[[c]] \leftarrow \mathsf{BinaryLessThanPublic}(\{[[r_i]]\}_{i=1}^k, \{\langle z_i, z_i \rangle\}_{i=1}^k)$
10: $[[z'_1]] \leftarrow 1 - [[c]]$
11: **for each** i in $\{2, \ldots, k+1\}$ **do**
12: $\quad [[z'_i]] \leftarrow [[r_{i-1}]] - 2z_{i-1}[[r_{i-1}]] + z_{i-1}$
13: **end for**
14: $\{[[Z_i]]\}_{i=1}^{k+1} \leftarrow \mathsf{PrefixOr}(\{[[z'_i]]_{i=1}^{k+1}\})$
15: **for each** i in $\{1, \ldots, k\}$ **do**
16: $\quad [[t_i]] \leftarrow [[Z_i]] - (1 - z_i)[[r_i]]$
17: **end for**
18: $\{[[T_i]]\}_{i=1}^k \leftarrow \mathsf{PrefixOr}(\{[[t_i]]\}_{i=1}^k)$
19: $[[f_i]] \leftarrow z_i[[T_i]]$
20: $[[g_i]] \leftarrow [[r_i]] \cdot [[T_i]]$
21: $[[w]] \leftarrow \mathsf{BinaryLessThan}(\{[[f_i]]\}_{i=1}^k, \{[[g_i]]\}_{i=1}^k)$
22: $[[b_{\text{odd}}]] \leftarrow \sum_{i=1}^{k/2}(1 - [[T_{2i}]])$
23: $[[b_{\text{even}}]] \leftarrow \sum_{i=1}^{k/2-1}(1 - [[T_{2i+1}]])$
24: $[[b]] \leftarrow 1 + [[w]] \cdot ([[b_{\text{odd}}]] - [[b_{\text{even}}]]) + [[b_{\text{even}}]]$
25: $[2^{2t}] \leftarrow \mathsf{TableLookupWithFieldConversion}(\{i \mapsto 2^{2(i-1)}\}_{i=1}^{k/2+1}, [[b]])$
26: $[2^t] \leftarrow \mathsf{TableLookupWithFieldConversion}(\{i \mapsto 2^{i-1}\}_{i=1}^{k/2+1}, [[b]])$ (in parallel with Line 26)
27: **return** $[2^{2t}]$, $[2^t]$

Protocol 23: $[a/b]^{(f)} \leftarrow \mathsf{Divide}([a]^{(f)}, [b]^{(f)})$ (based on [13])

Require: $|a|, |b| < 2^{k-f-1}$ for even k, b strictly positive

1: $\theta \leftarrow 2\lceil \log_2(k/3.5) \rceil + 1$
2: $[2^{2t}], [2^t] \leftarrow \mathsf{NormalizerEvenExponent}([b]^{(f)})$
3: $[\bar{b}]^{(f)} \leftarrow \mathsf{Truncate}([2^{2t}][b]^{(f)}, k, k-f)$
4: $[\bar{b}^2]^{(f)} \leftarrow \mathsf{MultiplyFP}([\bar{b}]^{(f)}, [\bar{b}]^{(f)})$
5: $[\bar{w}_0]^{(f)} \leftarrow E_f(5.9430) - 10[\bar{b}]^{(f)} + 5[\bar{b}^2]^{(f)}$
6: $[w_0]^{(f)} \leftarrow \mathsf{Truncate}([2^{2t}][\bar{w}_0]^{(f)}, k+f+2, k-f)$
7: $[x_0]^{(f)} \leftarrow \mathsf{MultiplyFP}([a]^{(f)}, [w_0]^{(f)})$
8: $[y_0]^{(f)} \leftarrow E_f(1) - \mathsf{MultiplyFP}([b]^{(f)}, [w_0]^{(f)})$
9: **for each** t in $\{1, \ldots, \theta\}$ **do**
10: $\quad [x_t]^{(f)} \leftarrow \mathsf{MultiplyFP}([x_{t-1}]^{(f)}, E_f(1) + [y_{t-1}]^{(f)})$
11: $\quad [y_t]^{(f)} \leftarrow \mathsf{MultiplyFP}([y_{t-1}]^{(f)}, [y_{t-1}]^{(f)})$
12: **end for**
13: **return** $[x_\theta]^{(f)}$

Protocol 24: $[\sqrt{a}]^{(f)}, [1/\sqrt{a}]^{(f)} \leftarrow \mathsf{SqrtAndSqrtInverse}([a]^{(f)})$

Require: $|a| < 2^{k-f-1}$ for even integers k and f, a strictly positive

1: $\theta \leftarrow 2\lceil \log_2(k/3.5) \rceil$
2: $[2^{2t}], [2^t] \leftarrow \mathsf{NormalizerEvenExponent}([a]^{(f)})$
3: $[\bar{a}]^{(f)} \leftarrow \mathsf{Truncate}([2^{2t}][a]^{(f)}, k, k-f)$
4: $[\bar{a}^2]^{(f)} \leftarrow \mathsf{MultiplyFP}([\bar{a}]^{(f)}, [\bar{a}]^{(f)})$
5: $[\bar{w}_0]^{(f)} \leftarrow E_f(2.9581) - 4[\bar{a}]^{(f)} + 2[\bar{a}^2]^{(f)}$
6: $[w_0/2]^{(f)} \leftarrow \mathsf{Truncate}([2^t][\bar{w}_0]^{(f)}, k/2+f+2, (k-f)/2+1)$
7: $[x_0]^{(f)} \leftarrow \mathsf{MultiplyFP}([a]^{(f)}, 2[w_0/2]^{(f)})$
8: $[y_0]^{(f)} \leftarrow [w_0/2]^{(f)}$
9: **for each** t in $\{1, \ldots, \theta\}$ **do**
10: $\quad [z_t]^{(f)} \leftarrow E_f(1.5) - \mathsf{MultiplyFP}([x_{t-1}]^{(f)}, [y_{t-1}]^{(f)})$
11: $\quad [x_t]^{(f)} \leftarrow \mathsf{MultiplyFP}([z_t]^{(f)}, [x_{t-1}]^{(f)})$
12: $\quad [y_t]^{(f)} \leftarrow \mathsf{MultiplyFP}([z_t]^{(f)}, [y_{t-1}]^{(f)})$ (in parallel with Line 11)
13: **end for**
14: **return** $[x_\theta]^{(f)}$, $2[y_\theta]^{(f)}$

Supplementary Note 6: Our Choice of Base Primes

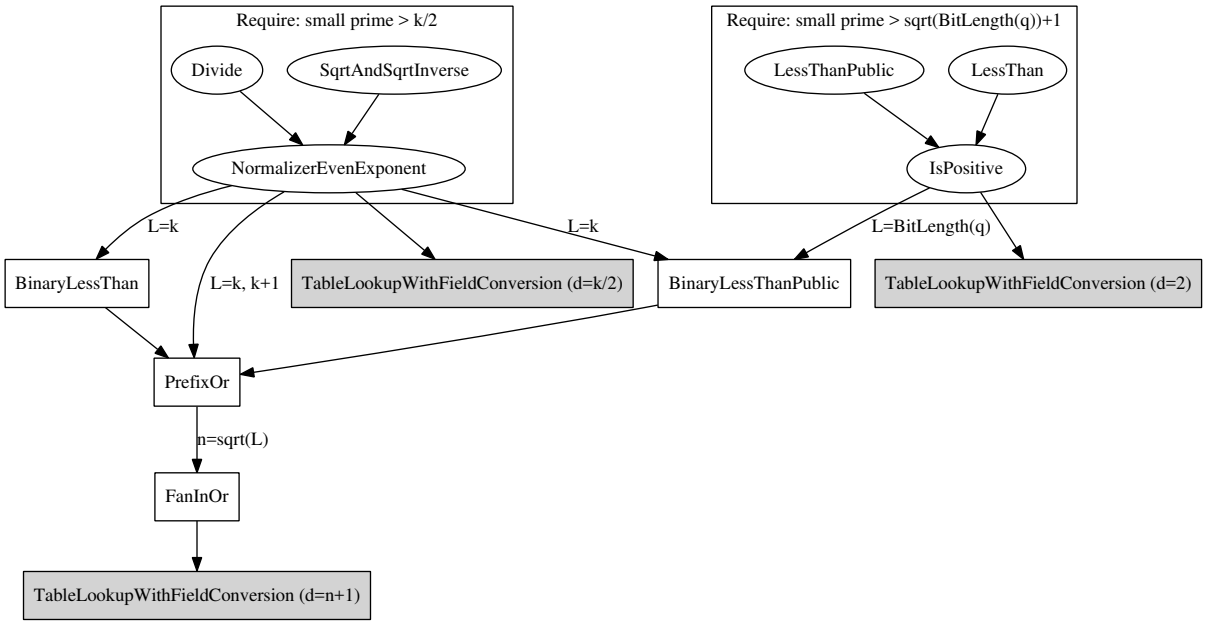
Here, we describe how we choose the size of finite fields (e.g., $\mathbb{Z}_q$) for secret sharing.

Choosing the primary field $\mathbb{Z}_q$. Recall first that our supported data range $\mathcal{D} \subset \mathbb{R}$ is defined to be $[-2^{k-f-1}, 2^{k-f-1})$, where k is the total number of bits used for the fixed point representation and $f < k$ is the number of bits assigned to the fractional range. Also, κ is the statistical security parameter for the subroutines that provide statistical security guarantees (i.e., Truncate and NormalizerEvenExponent). Whenever we multiply two secret-shared FP values, we obtain the value $[x]^{(2f)}$ for some $x \in \mathcal{D}$. This means q must have at least $k + f$ bits. Moreover, invoking Truncate subroutine on $[x]^{(2f)}$ to obtain $[x]^{(f)}$ requires us to generate a $(k + f + \kappa)$-bit random number (Line 1 of Truncate, with $b = k + f$), which lower bounds q at $2^{k+f+\kappa}$. We ensure that the maximum precision of a secret FP number in our protocol is $2f$ (by invoking Truncate when necessary), although one could use higher precisions at the expense of increasing q. In our setting, the highest lower bound on q is introduced by Line 6 of Divide where we invoke Truncate with $k + f + 2$. Thus, our final requirement for q is

$$q > k + f + \kappa + 2.$$

Choosing the auxiliary fields. For the small auxiliary field for bit operations, we choose two different base primes q_1 and q_2 for maximal efficiency. Supplementary Figure 3 (below) shows the call graph of subroutines that involve bit operations.

In our protocols, TableLookupWithFieldConversion imposes a lower bound on the small prime, as the size of the field must be at least as large as the number of entries in the table. When the secret shares are being constructed in the auxiliary field, we find all associated table lookups and choose the smallest prime that is larger than the largest table size. In summary, NormalizerEvenExponent,



Supplementary Figure 3: Call graph of subroutines that utilize the auxiliary finite field.

Nature Biotechnology: doi:10.1038/nbt.4108

which Divide and SqrtAndSqrtInverse depend on, uses

$$q_1 > k/2,$$

whereas IsPositive, which LessThanPublic and LessThan depend on, uses

$$q_2 > \sqrt{\mathsf{BitLength}(q)} + 1.$$

Note that q_2 is often much smaller than q_1, and thus our use of two auxiliary fields leads to a substantial improvement in efficiency compared to a one-size-fits-all approach.

In addition, we note that our use of NormalizerEvenExponent for Divide, as opposed to the more natural normalization routine without the requirement of the exponent being even, is intended for efficiency. If we were to take the alternative approach, the size of the table lookup at the end of the procedure becomes k instead of $k/2$. This doubles the size of the small field used for the entire protocol execution, which in turn, effectively doubles the amount of communication.

Parameters for experiments. The parameter setting we used for our benchmark experiments is as follows: $k = 60$, $f = 30$, $\kappa = 64$, $q = 2^{160} - 47$, $q_1 = 31$, and $q_2 = 17$. Base primes q, q_1, and q_2 are chosen to satisfy the requirements above. The data range parameters k and f naturally depend on the input data dimensions and the required level of precision. Our choice was sufficient to obtain accurate results for the benchmark data sets. While larger data may require higher precision, we note that even doubling the precision (e.g., $k = 120$ and $f = 60$) increases the size of each field element (the bit length of q) by only roughly 40%, and thus maintains the practical feasibility of our protocol.

Supplementary Note 7: Further Optimization with Shared Random Streams

To further improve performance, we use a pseudorandom generator (PRG) to generate the random bits (used primarily for secret sharing and Beaver partitioning). At a high level, a PRG takes as input a short uniformly random seed (e.g., 128-bits) and outputs a long stream of random-looking bits (e.g., 2^{64} bits). The security guarantee is that no "efficient" adversary (i.e., a polynomial-time algorithm) is able to distinguish the output of the PRG from a uniformly random sequence of random bits. Using a PRG to derive a random stream of bits allows us to significantly reduce the communication needed in the protocol. For instance, instead of sending a party a long stream of uniformly random bits, one can instead send a short PRG seed and have the receiving party derive the random bits by evaluating the PRG.

This optimization significantly improves the performance of ShareSecret and BeaverPartition. Protocols 25 and 26 show the improved versions of the respective protocols using shared random streams, which are denoted as

$$\text{PRG}_{\mathcal{P}} := \text{An instance of pseudorandom generator shared among the parties in } \mathcal{P}.$$

In ShareSecretSharedPRG, the data owner (either SP or CP_0) communicates with one of the main computing parties only, since the other party can obtain its share from the shared random stream. Next, in BeaverPartitionSharedPRG, we replace CP_0's sampling of r with an implicit sampling procedure, where CP_1 and CP_2 randomly chooses the respective shares $[r]_1$ and $[r]_2$ first and CP_0 retroactively recovers r by adding the two shares obtained offline via the shared random streams. Since $[r]_1$ and $[r]_2$ are uniformly random, we maintain the property that r is uniformly random. Note CP_0 does not communicate with CP_1 or CP_2 in this modified protocol—neither online nor offline (precomputation). We summarize the improved communication complexities of our method for previously described settings in Supplementary Table 8.

Protocol 25: $[x] \leftarrow \mathsf{ShareSecretSharedPRG}(\langle \perp, \perp, \perp, x \rangle)$ (or $\langle \perp, \perp, x \rangle$)

1: SP (or CP_0): Compute $r \in \mathbb{Z}_q$ using $\text{PRG}_{\mathsf{SP},\mathsf{CP}_1}$ (or $\text{PRG}_{\mathsf{CP}_0,\mathsf{CP}_1}$)
2: CP_1: Compute $r \in \mathbb{Z}_q$ using $\text{PRG}_{\mathsf{SP},\mathsf{CP}_1}$ (or $\text{PRG}_{\mathsf{CP}_0,\mathsf{CP}_1}$)
3: SP (or CP_0): Send $x - r$ to CP_2
4: **return** $\langle r, x - r \rangle$

Protocol 26: $\langle x - r, x - r \rangle, \langle [r]_1, [r]_2, r \rangle \leftarrow \mathsf{BeaverPartitionSharedPRG}([x])$

1: $\mathsf{CP}_{0,1}$: Compute $[r]_1 \in \mathbb{Z}_q$ using $\text{PRG}_{\mathsf{CP}_0,\mathsf{CP}_1}$
2: $\mathsf{CP}_{0,2}$: Compute $[r]_2 \in \mathbb{Z}_q$ using $\text{PRG}_{\mathsf{CP}_0,\mathsf{CP}_2}$
3: CP_0: Calculate $r \leftarrow [r]_1 + [r]_2$
4: $\langle x - r, x - r \rangle \leftarrow \mathsf{ReconstructSecret}([x] - [r])$
5: **return** $\langle x - r, x - r \rangle, \langle [r]_1, [r]_2, r \rangle$

Using PRGs to construct the random streams significantly reduces the bandwidth of our overall protocol. In particular, the initial Beaver partitioning of the input genotype matrix (a matrix with a million rows and a million columns) no longer requires CP_0 to send an equally-large random matrix to each of the main computing parties (CP_1 and CP_2). In fact, because the communication needed for matrix multiplication in our Beaver partitioning framework is proportional to the size of

Nature Biotechnology: doi:10.1038/nbt.4108

	Arith. circuit (Supp. Table 4)	**Matrix mult. (Supp. Table 5)**	**Power iteration (Supp. Table 6)**
Rounds (online)	1	1	$2T$
Bandwidth (online), $\mathsf{CP}_1 \leftrightarrow \mathsf{CP}_2$	k	$d_1d_2 + d_2d_3$	$d_1d_2 + d_3T(d_1 + d_2)$
Bandwidth (offline), $\mathsf{CP}_0 \rightarrow \mathsf{CP}_{1/2}$	$m + \tilde{T}$	d_1d_3	$d_3T(d_1 + d_2)$

Supplementary Table 8: Communication complexity of our improved method, combining our Beaver partitioning method and shared random streams, for previously analyzed examples (Supplementary Tables 4–6). The arithmetic circuit has k inputs, m outputs, and linearization cost $\tilde{T}$ (Supplementary Note 3). Recall that $\tilde{T} = 0$ when the multiplicative depth of the arithmetic circuit is at most 1 (e.g., in the specific case of matrix multiplication). The second column considers computing the product of a d_1-by-d_2 matrix with a d_2-by-d_3 matrix. The third column considers power iteration on an initial d_2-by-d_3 matrix, where on each of T iterations, the matrix is left-multiplied by a d_1-by-d_2 matrix and then by its transpose (same matrix is used for every iteration).

the *output* matrix (Supplementary Table 8), this yields a considerable communication reduction if we ensure that all matrix multiplications involving the genotype matrix outputs a thin (but long) matrix or a short (but fat) matrix. In this way, we effectively reduce the communication complexity for Beaver partitioning the initial genotype matrix from quadratic to linear in the dimensions of the genotype matrix (which constitutes a *million-fold* improvement in many realistic scenarios).

Supplementary Note 8: Our Secure Linear Algebra Subroutines

Here, we describe the subroutines for linear algebraic operations that we developed and used to implement principal component analysis (PCA) in GWAS.

Notation. We extend our previous notation for secret sharing to matrices (which includes vectors and scalars as special cases). Typically, we will use bold-faced uppercase letters (e.g., $\mathbf{A}, \mathbf{B}$) to denote matrices and bold-faced lowercase letters (e.g., $\mathbf{u}, \mathbf{v}$) to denote vectors. For a matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$ and an index $1 \leq i \leq n$ we write $\mathbf{A}_{i:,:}$ to denote the submatrix corresponding to the bottom $n - i + 1$ rows of $\mathbf{A}$ (namely, the matrix consisting of rows $i, i+1, \ldots, n$ of $\mathbf{A}$). We write $\mathbf{A}_{:i,:}$ to denote the submatrix consisting of the top i rows of $\mathbf{A}$. Similarly, for a column index $1 \leq j \leq m$, we write $\mathbf{A}_{:,j:}$ to denote the submatrix consisting of the rightmost $m - j + 1$ columns of $\mathbf{A}$, and $\mathbf{A}_{:,:j}$ to denote the leftmost j columns of $\mathbf{A}$. In some cases, we also combine the two; for instance, we write $\mathbf{A}_{:i,:j}$ to denote the submatrix of $\mathbf{A}$ containing the first i rows and j columns of $\mathbf{A}$.

For a real-valued matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$, we define its secret sharing to be a secret-sharing of the fixed-point encoding of $\mathbf{A}$:

$$[\mathbf{A}]^{(f)} := \begin{bmatrix} [\mathbf{A}_{1,1}]^{(f)} & \cdots & [\mathbf{A}_{1,m}]^{(f)} \\ \vdots & \ddots & \vdots \\ [\mathbf{A}_{n,1}]^{(f)} & \cdots & [\mathbf{A}_{n,m}]^{(f)} \end{bmatrix}.$$

We also extend our previously-defined protocols and functions that operate on scalar values to operate component-wise on matrices. For instance, invoking ShareSecret or ShareSecretSharedPRG on a matrix is equivalent to invoking the protocol on each element in parallel so that every element is individually secret shared.

Secure arithmetic with matrices. Much of our previous discussion in Supplementary Note 2 naturally extends to arithmetic operations with matrices. For example, affine functions over secret-shared matrices can be performed non-interactively, since each element of the resulting matrix is an affine function over the secret-shared elements in the input. Secure multiplication of two matrices is efficiently handled by our generalized Beaver partitioning method (Supplementary Note 3, Section 3.2). Furthermore, if the same matrix is used across multiple multiplications, its Beaver-partitioned data can be reused (Supplementary Note 3, Section 3.3). Note that to simplify the protocol description, we do not explicitly describe how the variable reuse optimization is applied in our protocol.

Protocols. We now describe the subroutines used for PCA. All of the protocols described here implement secure computations over secret-shared real values (represented as fixed-point values, as described in Supplementary Note 5).

- Householder$([\mathbf{x}]^{(f)}) \to [\mathbf{v}]^{(f)}$: On input a secret-shared vector $[\mathbf{x}]^{(f)}$ where $\mathbf{x} \in \mathbb{R}^d$, the Householder protocol (Protocol 27) outputs a share $[\mathbf{v}]^{(f)}$ of a unit vector $\mathbf{v} \in \mathbb{R}^d$ such that the reflection of x about the hyperplane associated with $\mathbf{v}$ (this is also known as the "Householder reflection") yields a vector that is a multiple of the elementary vector $\mathbf{e}_1$ (i.e., only the first element is non-zero). In other words, $(1 - \mathbf{v}\mathbf{v}^T)\mathbf{x} = \alpha \mathbf{e}_1$ for some $\alpha \in \mathbb{R}$.
- QRFactorizeSquare$([\mathbf{A}]^{(f)}) \to ([\mathbf{Q}]^{(f)}, [\mathbf{R}]^{(f)})$: On input a secret-shared matrix $[\mathbf{A}]^{(f)}$ where $\mathbf{A} \in \mathbb{R}^{d \times d}$, the QRFactorizeSquare protocol (Protocol 28) securely executes a standard QR

Protocol 27: $[\mathbf{v}]^{(f)} \leftarrow$Householder$([\mathbf{x}]^{(f)})$

Require: $\mathbf{x} \in \mathbb{R}^d$

1: $[c] \leftarrow$ IsPositive$([\mathbf{x}_1]^{(f)})$
2: $[\text{sign}(\mathbf{x}_1)] \leftarrow 2[c] - 1$
3: $[\|\mathbf{x}\|^2]^{(f)} \leftarrow$ Truncate$([\mathbf{x}^T]^{(f)}[\mathbf{x}]^{(f)}, k+f, f)$
4: $[\|\mathbf{x}\|]^{(f)}, [1/\|\mathbf{x}\|]^{(f)} \leftarrow$ SqrtAndSqrtInverse$([\|\mathbf{x}\|^2]^{(f)})$
5: $[\mathbf{u}]^{(f)} \leftarrow [\mathbf{x}]^{(f)} + [\text{sign}(\mathbf{x}_1)][\|\mathbf{x}\|]^{(f)} e_1$.
6: $[\|\mathbf{u}\|^2]^{(f)} \leftarrow$ Truncate$([\mathbf{u}^T]^{(f)}[\mathbf{u}]^{(f)}, k+f, f)$
7: $[\|\mathbf{u}\|]^{(f)}, [1/\|\mathbf{u}\|]^{(f)} \leftarrow$ SqrtAndSqrtInverse$([\|\mathbf{u}\|^2]^{(f)})$
8: $[\mathbf{v}]^{(f)} \leftarrow$ Truncate$([1/\|\mathbf{u}\|]^{(f)}[\mathbf{u}]^{(f)}, k+f, f)$.
9: **return** $[\mathbf{v}]^{(f)}$

factorization algorithm, which applies $d-1$ Householder transformations $\mathbf{Q}_1, \ldots, \mathbf{Q}_{d-1}$ to $\mathbf{A}$ to obtain an upper-triangular matrix $\mathbf{R} = \mathbf{Q}_{d-1} \cdots \mathbf{Q}_1 \mathbf{A}$. Setting $\mathbf{Q} := \mathbf{Q}_1 \cdots \mathbf{Q}_{d-1}$ yields the QR factorization $\mathbf{A} = \mathbf{QR}$. The output of the protocol are secret shares $[\mathbf{Q}]^{(f)}$ and $[\mathbf{R}]^{(f)}$ of $\mathbf{Q}$ and $\mathbf{R}$, respectively. We also present a modified protocol QRFactorizeRectangle (Protocol 29) for tall and skinny matrices $\mathbf{A}$ where the number of rows d is large enough that storing a d-by-d matrix in memory (as in QRFactorizeSquare) is not feasible. This is the case in our GWAS application.

Protocol 28: $[\mathbf{Q}]^{(f)}, [\mathbf{R}]^{(f)} \leftarrow$QRFactorizeSquare$([\mathbf{A}]^{(f)})$

Require: $\mathbf{A} \in \mathbb{R}^{d \times d}$

1: $[\mathbf{Q}]^{(f)} \leftarrow \langle \mathbf{0}_{d\times d}, E_f(\mathbf{I}_{d\times d}) \rangle$, where $\mathbf{I}_{d\times d} \in \mathbb{R}^{d\times d}$ is the identity matrix of dimension d
2: $[\mathbf{R}]^{(f)} \leftarrow \langle \mathbf{0}_{d\times d}, \mathbf{0}_{d\times d} \rangle$
3: $[\mathbf{A}^{(1)}]^{(f)} \leftarrow [\mathbf{A}]^{(f)}$
4: **for** $i = 1, \ldots, d-1$ **do**
5: $\quad [\mathbf{v}]^{(f)} \leftarrow$ Householder$([\mathbf{A}^{(i)}_{:,1}]^{(f)})$
6:
7: $\quad$ /* Householder-transform $\mathbf{A}^{(i)}$ and $\mathbf{Q}_{:,i:}$ */
8: $\quad [\mathbf{v}^T \mathbf{A}^{(i)}]^{(f)} \leftarrow$ Truncate$([\mathbf{v}^T]^{(f)}[\mathbf{A}^{(i)}]^{(f)}, k+f, f)$ (in parallel with Line 9)
9: $\quad [\mathbf{Q}_{:,i:} v]^{(f)} \leftarrow$ Truncate$([\mathbf{Q}_{:,i:}]^{(f)}[\mathbf{v}]^{(f)}, k+f, f)$
10: $\quad [\mathbf{v}\mathbf{v}^T \mathbf{A}^{(i)}]^{(f)} \leftarrow$ Truncate$([\mathbf{v}]^{(f)}[\mathbf{v}^T \mathbf{A}^{(i)}]^{(f)}, k+f, f)$ (in parallel with Line 11)
11: $\quad [\mathbf{Q}_{:,i:} v v^T]^{(f)} \leftarrow$ Truncate$([\mathbf{Q}_{:,i:}\mathbf{v}]^{(f)}[\mathbf{v}^T]^{(f)}, k+f, f)$
12: $\quad [\mathbf{A}^{(i)}]^{(f)} \leftarrow [\mathbf{A}^{(i)}]^{(f)} - 2[\mathbf{v}\mathbf{v}^T \mathbf{A}^{(i)}]^{(f)}$
13: $\quad [\mathbf{Q}_{:,i:}]^{(f)} \leftarrow [\mathbf{Q}_{:,i:}]^{(f)} - 2[\mathbf{Q}_{:,i:}\mathbf{v}\mathbf{v}^T]^{(f)}$
14:
15: $\quad [\mathbf{R}_{i,i:}]^{(f)} \leftarrow [\mathbf{A}^{(i)}_{1,:}]^{(f)}$
16: $\quad [\mathbf{A}^{(i+1)}]^{(f)} \leftarrow [\mathbf{A}^{(i)}_{2:,2:}]^{(f)}$
17: **end for**
18: $[\mathbf{R}_{d,d}]^{(f)} \leftarrow [\mathbf{A}^{(d)}_{1,1}]^{(f)}$
19: **return** $[\mathbf{Q}]^{(f)}$ and $[\mathbf{R}]^{(f)}$

- Tridiagonalize$([\mathbf{A}]^{(f)}) \to ([\mathbf{Q}]^{(f)}, [\mathbf{T}]^{(f)})$: On input a secret-shared symmetric matrix $[\mathbf{A}]^{(f)}$ where $\mathbf{A} \in \mathbb{R}^{d\times d}$, the Tridiagonalize protocol (Protocol 30) computes matrices $\mathbf{Q}$ and $\mathbf{T}$ such that $\mathbf{T} = \mathbf{QAQ}^T$ and $\mathbf{T}$ is *tridiagonal* (namely, it only has non-zero elements along

Protocol 29: $[\mathbf{Q}]^{(f)}, [\mathbf{R}]^{(f)} \leftarrow$QRFactorizeRectangle$([\mathbf{A}]^{(f)})$

Require: $\mathbf{A} \in \mathbb{R}^{d \times r}$ where $r \ll d$, and $d \times d$ matrix is too large to fit in memory

$[\mathbf{R}]^{(f)} \leftarrow \langle \mathbf{0}_{r \times r}, \mathbf{0}_{r \times r} \rangle$
$[\mathbf{A}^{(1)}]^{(f)} \leftarrow [\mathbf{A}]^{(f)}$
for $i = 1, \ldots, r$ **do**
    $[\mathbf{v}^{(i)}]^{(f)} \leftarrow$ Householder$([\mathbf{A}^{(i)}_{:,1}]^{(f)})$

    /* Householder-transform $\mathbf{A}^{(i)}$ */
    $[(\mathbf{v}^{(i)})^T A^{(i)}]^{(f)} \leftarrow$ Truncate$([(\mathbf{v}^{(i)})^T]^{(f)}[\mathbf{A}^{(i)}]^{(f)}, k+f, f)$
    $[\mathbf{v}^{(i)}(\mathbf{v}^{(i)})^T \mathbf{A}^{(i)}]^{(f)} \leftarrow$ Truncate$([\mathbf{v}^{(i)}]^{(f)}[(\mathbf{v}^{(i)})^T \mathbf{A}^{(i)}]^{(f)}, k+f, f)$
    $[\mathbf{A}^{(i)}]^{(f)} \leftarrow [\mathbf{A}^{(i)}]^{(f)} - 2[\mathbf{v}\mathbf{v}^T \mathbf{A}^{(i)}]^{(f)}$

    $[\mathbf{R}_{i,i:}]^{(f)} \leftarrow [\mathbf{A}^{(i)}_{1,:}]^{(f)}$
    $[\mathbf{A}^{(i+1)}]^{(f)} \leftarrow [\mathbf{A}^{(i)}_{2:,2:}]^{(f)}$
end for
$[\mathbf{Q}]^{(f)} \leftarrow \langle \mathbf{0}_{d \times r}, E_f(\mathbf{I}_{d \times r}) \rangle$
for $i = r, \ldots, 1$ **do**
    /* Householder-transform $\mathbf{Q}_{i:,:}$ */
    $[(\mathbf{v}^{(i)})^T \mathbf{Q}_{i:,:}]^{(f)} \leftarrow$ Truncate$([(\mathbf{v}^{(i)})^T]^{(f)}[\mathbf{Q}_{i:,:}]^{(f)}, k+f, f)$
    $[\mathbf{v}^{(i)}(\mathbf{v}^{(i)})^T \mathbf{Q}_{i:,:}]^{(f)} \leftarrow$ Truncate$([\mathbf{v}^{(i)}]^{(f)}[(\mathbf{v}^{(i)})^T \mathbf{Q}_{i:,:}]^{(f)}, k+f, f)$
    $[\mathbf{Q}_{i:,:}]^{(f)} \leftarrow [\mathbf{Q}_{i:,:}]^{(f)} - 2[\mathbf{v}\mathbf{v}^T \mathbf{Q}_{i:,:}]^{(f)}$
end for
return $[\mathbf{Q}]^{(f)}, [\mathbf{R}]^{(f)}$

the diagonal, subdiagonal and superdiagonal). Similar to the QR factorization algorithm, tridiagonalization is achieved by iteratively applying Householder transformation to portions of $\mathbf{A}$ so that any element outside of the tridiagonal region becomes zero. The output of the algorithm is secret shares $[\mathbf{Q}]^{(f)}$ and $[\mathbf{T}]^{(f)}$ of $\mathbf{Q}$ and $\mathbf{T}$, respectively.

- Eigendecompose$([\mathbf{A}]^{(f)}) \rightarrow ([\mathbf{Q}]^{(f)}, [\mathbf{L}]^{(f)})$: On input a secret-shared symmetric matrix $[\mathbf{A}]^{(f)}$ where $\mathbf{A} \in \mathbb{R}^{d \times d}$, the Eigendecompose protocol applies the QR algorithm for eigendecomposition to $\mathbf{A}$ [19]. First, $\mathbf{A}$ is transformed using Tridiagonalize to improve the convergence of the subsequent steps. Next, a *QR iteration* is iteratively applied to the matrix until convergence, where each iteration consists of first QR factorizing the matrix as $\mathbf{QR}$, then updating the matrix as $\mathbf{RQ}$. Repeating this procedure pushes the last diagonal element towards the smallest eigenvalue of A, and upon convergence, the last row and column are deleted to repeat this procedure on the remaining eigenvalues until all of them are obtained. Notably, the eigenvalues revealed by this procedure are already sorted. We use the standard technique of "shifting" the matrix by the last diagonal element before and after each QR factorization to achieve cubic convergence [20]. Instead of testing for convergence, which may leak information, we fix the number of iterations per eigenvalue to a pre-determined value. In our experiments, five QR iterations per eigenvalue were sufficient to obtain accurate results.

Protocol 30: $[\mathbf{Q}]^{(f)}, [\mathbf{T}]^{(f)} \leftarrow \mathsf{Tridiagonalize}([\mathbf{A}]^{(f)})$ (based on [18])

Require: $\mathbf{A} \in \mathbb{R}^{d \times d}$ symmetric

1: $[\mathbf{Q}]^{(f)} \leftarrow \langle \mathbf{0}_{d\times d}, E_f(\mathbf{I}_{d\times d})\rangle$
2: $[\mathbf{T}]^{(f)} \leftarrow \langle \mathbf{0}_{d\times d}, \mathbf{0}_{d\times d}\rangle$
3: $[\mathbf{A}^{(1)}]^{(f)} \leftarrow [\mathbf{A}]^{(f)}$
4: **for** $i = 1, \ldots, d-2$ **do**
5: $\quad [\mathbf{v}]^{(f)} \leftarrow \mathsf{Householder}([\mathbf{A}^{(i)}_{2:,1}]^{(f)})$
6: $\quad [\mathbf{v}\mathbf{v}^T]^{(f)} \leftarrow \mathsf{Truncate}([\mathbf{v}]^{(f)}[\mathbf{v}^T]^{(f)}, k+f, f)$
7: $\quad [\mathbf{P}]^{(f)} \leftarrow \begin{bmatrix} \langle 0,1\rangle & \langle 0,0\rangle \\ \langle 0,0\rangle & \mathbf{I}_{(d-i)\times(d-i)} - 2[\mathbf{v}\mathbf{v}^T]^{(f)} \end{bmatrix}$
8: $\quad [\mathbf{P}\mathbf{A}^{(i)}]^{(f)} \leftarrow \mathsf{Truncate}([\mathbf{P}]^{(f)}[\mathbf{A}^{(i)}]^{(f)}, k+f, f)$ (in parallel with Line 9)
9: $\quad [\mathbf{P}\mathbf{Q}_{i:,:}]^{(f)} \leftarrow \mathsf{Truncate}([\mathbf{P}]^{(f)}[\mathbf{Q}_{i:,:}]^{(f)}, k+f, f)$
10: $\quad [\mathbf{P}\mathbf{A}^{(i)}\mathbf{P}^T]^{(f)} \leftarrow \mathsf{Truncate}([\mathbf{P}\mathbf{A}^{(i)}]^{(f)}[\mathbf{P}^T]^{(f)}, k+f, f)$
11: $\quad [\mathbf{A}^{(i)}]^{(f)} \leftarrow [\mathbf{P}\mathbf{A}^{(i)}\mathbf{P}^T]^{(f)}$
12: $\quad [\mathbf{Q}_{i:,:}]^{(f)} \leftarrow [\mathbf{P}\mathbf{Q}_{i:,:}]^{(f)}$
13: $\quad [\mathbf{T}_{i,i}]^{(f)} \leftarrow [\mathbf{A}^{(i)}_{1,1}]^{(f)}$, $[\mathbf{T}_{i,i+1}]^{(f)} \leftarrow [\mathbf{A}^{(i)}_{1,2}]^{(f)}$, and $[\mathbf{T}_{i+1,i}]^{(f)} \leftarrow [\mathbf{A}^{(i)}_{2,1}]^{(f)}$
14: $\quad [\mathbf{A}^{(i+1)}]^{(f)} \leftarrow [\mathbf{A}^{(i)}_{2:,2:}]^{(f)}$
15: **end for**
16: $[\mathbf{T}_{d-1:,d-1:}]^{(f)} \leftarrow [\mathbf{A}^{(d-1)}]^{(f)}$
17: **return** $[\mathbf{Q}]^{(f)}, [\mathbf{T}]^{(f)}$

Protocol 31: $[\mathbf{Q}]^{(f)}, [\mathbf{L}]^{(f)} \leftarrow \mathsf{Eigendecompose}([\mathbf{A}]^{(f)})$

Require: $A \in \mathbb{R}^{d \times d}$ symmetric, number of QR iterations per eigenvalue ω

1: $[\mathbf{Q}]^{(f)}, [\mathbf{T}]^{(f)} \leftarrow \mathsf{Tridiagonalize}([A]^{(f)})$
2: $[\mathbf{Q}]^{(f)} \leftarrow [\mathbf{Q}^T]^{(f)}$
3: $[\mathbf{L}]^{(f)} \leftarrow [\mathbf{T}]^{(f)}$
4: **for** $i = d \ldots 1$ **do**
5: $\quad$ **for** $j = 1 \ldots \omega$ **do**
6: $\quad\quad [\mu]^{(f)} \leftarrow [\mathbf{L}_{i,i}]^{(f)}$
7: $\quad\quad [\mathbf{Q}']^{(f)}, [\mathbf{R}']^{(f)} \leftarrow \mathsf{QRFactorizeSquare}([\mathbf{L}_{:i,:i}]^{(f)} - [\mu]^{(f)}\mathbf{I}_{i\times i})$
8: $\quad\quad [\mathbf{R}'\mathbf{Q}']^{(f)} \leftarrow \mathsf{Truncate}([\mathbf{R}']^{(f)}[Q']^{(f)}, k+f, f)$
9: $\quad\quad [\mathbf{L}_{:i,:i}]^{(f)} \leftarrow [\mathbf{R}'\mathbf{Q}']^{(f)} + [\mu]^{(f)}\mathbf{I}_{i\times i}$
10: $\quad\quad [\mathbf{Q}_{:,:i}\mathbf{Q}']^{(f)} \leftarrow \mathsf{Truncate}([\mathbf{Q}_{:,:i}]^{(f)}[\mathbf{Q}']^{(f)}, k+f, f)$
11: $\quad\quad [\mathbf{Q}_{:,:i}]^{(f)} \leftarrow [\mathbf{Q}_{:,:i}\mathbf{Q}']^{(f)}$
12: $\quad$ **end for**
13: **end for**
14: **return** $[\mathbf{Q}]^{(f)}$, $[\mathbf{L}]^{(f)}$

Supplementary Note 9: Our Secure GWAS Protocol

Now we describe how we build a secure and efficient GWAS protocol using the tools described in the previous Supplementary Notes. We work in the crowdsourcing scenario where there are n study participants, denoted $\mathsf{SP}_1, \ldots, \mathsf{SP}_n$, and m candidate SNPs to be tested for association. For simplicity, we assume each SP owns the data of a single individual. In addition, we assume that the mapping from SNP indices $\{1, \ldots, m\}$ to known SNP identifiers (e.g., rsids) is fixed and public. The individuals' genotypes at each position are of course hidden.

9.1 Input Data

We represent the input data owned by each SP_i as follows:

- $\mathbf{g}_i^{\mathsf{AA}}, \mathbf{g}_i^{\mathsf{Aa}}, \mathbf{g}_i^{\mathsf{aa}} \in \{0,1\}^m$: $\mathbf{g}_{ij}^{\mathsf{AA}}, \mathbf{g}_{ij}^{\mathsf{Aa}}, \mathbf{g}_{ij}^{\mathsf{aa}}$ represents whether the j^{th} SNP of SP_i is homozygous-reference allele, heterozygous, homozygous-alternative allele, respectively
- $\mathbf{h}_i \in \{0,1\}^m$: $\mathbf{h}_{ij}$ represents whether the j^{th} SNP of SP_i is missing
- $\mathbf{c}_i \in \{0,1\}^\ell$: covariate features (e.g., age group membership) for SP_i
- $y_i \in \{0,1\}$: phenotype of interest (e.g., disease status) for SP_i

We let $\mathbf{g}_{ij}^{\mathsf{AA}} = \mathbf{g}_{ij}^{\mathsf{Aa}} = \mathbf{g}_{ij}^{\mathsf{aa}} = 0$ if the genotype is missing (i.e., $\mathbf{h}_{ij} = 1$). In addition, *minor allele dosages* $\mathbf{x}_i \in \{0,1,2\}^m$ are defined as

$$\mathbf{x}_i := \mathbf{g}_i^{\mathsf{Aa}} + 2 \cdot \mathbf{g}_i^{\mathsf{aa}},$$

which can be easily computed from the above data. We use a one-hot encoding of genotypes with $\mathbf{g}_i^{\mathsf{AA}}$, $\mathbf{g}_i^{\mathsf{Aa}}$, and $\mathbf{g}_i^{\mathsf{aa}}$ in order to obtain the separate counts for AA, Aa, and aa for each SNP, which are needed for the quality control procedure (Hardy-Weinberg equilibrium and heterozygosity filters). Note that while computing $\mathbf{x}_i$ from $\mathbf{g}_i^{\mathsf{AA}}$, $\mathbf{g}_i^{\mathsf{Aa}}$, and $\mathbf{g}_i^{\mathsf{aa}}$ is easy, the reverse is quite expensive due to the need to perform several equality tests in order to extract each genotype. For a similar reason, missing genotypes are encoded in a separate vector $\mathbf{h}_i$ as opposed to using a sentinel value in the genotype vector, which would also require equality tests to extract the information.

While all numbers in our data are binary-valued, it is straightforward to incorporate continuous values using the fixed-point representation (Supplementary Note 5). For instance, we can incorporate imputed genotypes by assigning probabilities (as FP values) to the corresponding entries in $\mathbf{g}^{\mathsf{AA}}$, $\mathbf{g}^{\mathsf{Aa}}$, and $\mathbf{g}^{\mathsf{aa}}$. Covariate features and the phenotype encodings can also be continuous.

9.2 Initial Data Sharing

During the initial data sharing phase, each SP_i samples a random seed for a PRG and sends it to CP_1 over a secure channel. This initializes $\text{PRG}_{\mathsf{CP}_1,\mathsf{SP}_i}$. Next, SP invokes $\mathsf{ShareSecretSharedPRG}$ (Protocol 25) to securely share its data with CP_1 and CP_2. Note that SP transfers data to only CP_2, since CP_1 non-interactively obtains its shares from $\text{PRG}_{\mathsf{CP}_1,\mathsf{SP}_i}$. After all n SPs have shared their data, CP_1 and CP_2 execute $\mathsf{BeaverPartitionSharedPRG}$ (Protocol 26) on the pooled data to prepare for the main computation phase. While this step requires communication bandwidth equal to the total size of input, it can be aggregated and transferred in a single batch. Thus, at the scale of tens of terabytes (when working with millions of genomes), we expect physically shipping hard drives to be a viable alternative for this step that may be more efficient than online transfer.

Nature Biotechnology: doi:10.1038/nbt.4108

At the end of this procedure, the computing parties CP_1 and CP_2 have access to the shares $[\mathbf{g}_i^{\mathsf{AA}}]$, $[\mathbf{g}_i^{\mathsf{Aa}}]$, $[\mathbf{g}_i^{\mathsf{aa}}]$, $[\mathbf{h}_i]$, $[\mathbf{c}_i]$, and $[y_i]$, as well as their Beaver partitions for all $i \in \{1, \ldots, n\}$. We also assume CPs have non-interactively computed the minor allele dosages

$$[\mathbf{x}_i] \leftarrow [\mathbf{g}_i^{\mathsf{Aa}}] + 2[\mathbf{g}_i^{\mathsf{aa}}]$$

for all $i \in \{1, \ldots, n\}$. Note that the Beaver partitions of $[\mathbf{x}_i]$ can be obtained for free by simply taking a linear combination of the Beaver partitions of $[\mathbf{g}_i^{\mathsf{Aa}}]$ and $[\mathbf{g}_i^{\mathsf{aa}}]$.

9.3 Phase 1: Quality Control

The first phase of the main GWAS computation includes common quality control filters for GWAS. In the following, we provide the list of filters we implemented. We write UB and LB to denote an upper bound and a lower bound, respectively, which take on different values for each filter. We assume that these bounds are public and fixed.

- Heterozygosity of individual i: $\mathsf{LB} \leq \frac{\sum_{j=1}^{m} \mathbf{g}_{ij}^{\mathsf{Aa}}}{m - \sum_{j=1}^{m} \mathbf{h}_{ij}} < \mathsf{UB}$
- Genotype missing rate of individual i: $\frac{\sum_{j=1}^{m} \mathbf{h}_{ij}}{m} < \mathsf{UB}$.
- Minor allele frequency (MAF) of SNP j: $\mathsf{LB} \leq \frac{\sum_{i=1}^{n} \mathbf{x}_{ij}}{2(n - \sum_{i=1}^{n} \mathbf{h}_{ij})} < \mathsf{UB}$
- Genotype missing rate of SNP j: $\frac{\sum_{i=1}^{n} \mathbf{h}_{ij}}{n} < \mathsf{UB}$
- Hardy-Weinberg equilibrium (HWE) χ^2 test statistic of SNP j (control cohort-only):

$$\sum_{t \in \{\mathsf{AA},\mathsf{Aa},\mathsf{aa}\}} \frac{(O_j^t - E_j^t)^2}{E_j^t} < \mathsf{UB},$$

where

$$O_j^t := \sum_{i=1}^{n} (1 - y_i) \mathbf{g}_{ij}^t, \forall t \in \{\mathsf{AA}, \mathsf{Aa}, \mathsf{aa}\},$$

$$E_j^{\mathsf{AA}} := \alpha_j^2 \left(n - \sum_{i=1}^{n} y_i \right) \quad E_j^{\mathsf{Aa}} := 2\alpha_j(1-\alpha_j) \left(n - \sum_{i=1}^{n} y_i \right) \quad E_j^{\mathsf{aa}} := (1-\alpha_j)^2 \left(n - \sum_{i=1}^{n} y_i \right),$$

and

$$\alpha_j := \frac{\sum_{i=1}^{n} y_i \mathbf{x}_{ij}}{2(n - \sum_{i=1}^{n} y_i \mathbf{h}_{ij})}.$$

After the CPs compute each of the above quantities over the secret-shared values, they compare each quantity against the (public) thresholds using the $\mathsf{LessThan}$ or $\mathsf{LessThanPublic}$ protocols (Protocols 20 and 21). For all filters except for HWE, we multiply the thresholds with the denominator of the term being compared to avoid invoking the relatively more expensive Divide protocol. This reduces the required computation (other than comparisons) to only affine functions over the secret shares, which can be computed non-interactively.

The HWE filter is the most complex among the quality control filters. First, the numerator and denominator of α_j are separately computed for all j, which is a depth-one computation with an overall output size $2m$. Given this result, we use m parallel invocations of Divide to securely compute α_j for all j. After computing the terms $(O_j^t - E_j^t)^2$ and E_j^t via secure multiplications, we

use three rounds of m parallel invocations of Divide (one for each $t \in \{\mathsf{AA}, \mathsf{Aa}, \mathsf{aa}\}$) to calculate the ratios $(O_j^t - E_j^t)^2/E_j^t$. Lastly, the results are added for each SNP to obtain the HWE test statistics, which are then compared with the threshold.

After the CPs have securely evaluated each of the above filters, the computing parties compute an AND over the filter outputs. The computing parties then publish their shares and reconstruct the binary inclusion/exclusion status for each individual or SNP (to be revealed also at the conclusion of GWAS along with the association statistics). We perform this step in advance to allow the CPs to directly filter the data sets for the subsequent steps, which cannot be done efficiently if the filtering results are kept hidden. As further explained at the end of this Supplementary Note, this information reveals only a single bit per SNP or per individual and therefore arguably poses a significantly smaller privacy risk than the publication of association statistics.

In practice, researchers may wish to apply some filters first and evaluate the remaining filters over the reduced data. For instance, if the data is pooled from different genotyping platforms, it may be desirable to filter out SNPs with high missing rates first in order to discard non-intersecting loci. In our experiments, we performed quality control in three stages: (i) locus missing rate filter, (ii) individual missing rate and heterozygosity filters, and (iii) locus HWE and MAF filters. The results from each stage was used to reduce the data set before proceeding to the next stage.

We use n_{qc} and m_{qc} to denote the number of individuals and SNPs passing all quality control filters, respectively.

9.4 Phase 2: Population Stratification Analysis

The next phase of the computation is population stratification analysis, where the goal is to obtain the top principal components of the genotype matrix via principal component analysis (PCA) to be included as covariates in the association tests.

SNP selection. The current standard practice is to perform PCA over a reduced set of SNPs that are largely independent from one another. Strong correlations among the SNPs, such as those arising from linkage disequilibrium (LD), can distort the PCA results and thus, need to be avoided [21]. In our protocol, we take the simplified approach of keeping only SNPs that are at least 100 Kb apart in order to minimize the impact of LD. Alternative approaches, such as directly computing pairwise correlations and filtering based on them, can be implemented at the expense of efficiency. Since annotations (e.g., genomic position) associated with each SNP in the input data is public, each party independently filters the SNPs according to the same procedure and obtains the reduced matrix non-interactively. Let m_{pca} be the resulting number of SNPs to be used for PCA.

Computing standardization parameters. We denote the filtered input matrix for PCA as $\mathbf{X} \in \{0,1,2\}^{n_{\text{qc}} \times m_{\text{pca}}}$, which contains minor allele dosages. The corresponding missingness data is also represented as a matrix $\mathbf{H} \in \{0,1\}^{n_{\text{qc}} \times m_{\text{pca}}}$.

To standardize the matrix before performing PCA, we follow previous work [22] to compute the mean μ_j and standard deviation σ_j of each SNP j as

$$\mu_j := \frac{1 + \sum_{i=1}^{n_{\text{qc}}} \mathbf{X}_{ij}}{2 + 2(n_{\text{qc}} - \sum_{i=1}^{n_{\text{qc}}} \mathbf{H}_{ij})},$$
$$\sigma_j := \sqrt{\mu_j(1-\mu_j)},$$

where the mean is computed over only the observed genotypes. Note μ_j is smoothed by adding a pseudocount for both allele types to avoid zero standard deviations.

To compute these parameters, the CPs first calculate the denominator and numerator of μ_j for all j, which are affine functions over the secret shares. Next, μ_j for all j are computed by m_{pca} parallel invocations of Divide. Then, μ_j and $1 - \mu_j$ are securely multiplied and provided as input to m_{pca} parallel invocations of SqrtAndSqrtInverse to obtain $1/\sigma_j$ for all j. We keep $1/\sigma_j$ instead of σ_j in order to standardize the genotypes by multiplication, and not by division, as

$$\tilde{\mathbf{X}}_{ij} := (1/\sigma_j) \cdot (\mathbf{X}_{ij} - \mu_j(1 - \mathbf{H}_{ij})),$$

where $\tilde{\mathbf{X}}$ denotes the standardize input matrix for PCA. Note we subtract the mean only where the genotype is not missing.

Implicit standardization. Explicitly constructing the standardized matrix $\tilde{\mathbf{X}}$ incurs a communication cost that scales quadratically in the dimension of the input data. This is because each element in $\tilde{\mathbf{X}}$ corresponds to an output gate that needs to be reconstructed. Instead, we adopt a lazy computation scheme for standardizing $\mathbf{X}$, where every occurrence of $\tilde{\mathbf{X}}$ in the following computation is replaced with

$$\tilde{\mathbf{X}} \mapsto (\mathbf{X} - \mathbf{HM})\mathbf{\Sigma}^{-1},$$

with

$$\mathbf{M} := \begin{bmatrix} \mu_1 & & 0 \\ & \ddots & \\ 0 & & \mu_{m_{\text{pca}}} \end{bmatrix} \quad \text{and} \quad \mathbf{\Sigma}^{-1} := \begin{bmatrix} 1/\sigma_1 & & 0 \\ & \ddots & \\ 0 & & 1/\sigma_{m_{\text{pca}}} \end{bmatrix}.$$

In our PCA protocol, $\tilde{\mathbf{X}}$ only appears in products where the resulting matrix is either tall-and-skinny or short-and-fat with the smaller dimension being a very small constant. After writing out each multiplication with the above substitution, we observe that we can evaluate matrix products involving $\tilde{\mathbf{X}}$ in one of two different ways:

$$\mathbf{A}\tilde{\mathbf{X}} \mapsto \mathbf{A}(\mathbf{X} - \mathbf{HM})\mathbf{\Sigma}^{-1} \mapsto (\mathbf{AX})\mathbf{\Sigma}^{-1} - ((\mathbf{AH})\mathbf{M})\mathbf{\Sigma}^{-1},$$
$$\tilde{\mathbf{X}}\mathbf{B} \mapsto (\mathbf{X} - \mathbf{HM})\mathbf{\Sigma}^{-1}\mathbf{B} \mapsto \mathbf{X}(\mathbf{\Sigma}^{-1}\mathbf{B}) - \mathbf{H}(\mathbf{M}(\mathbf{\Sigma}^{-1}\mathbf{B})),$$

where $\mathbf{A}$ is short-and-fat and $\mathbf{B}$ is tall-and-skinny. This way, each matrix multiplication involving $\mathbf{X}$ results in an intermediary matrix that has at least one very small dimension. This means that the overall communication bandwidth scales linear in m_{pca} and n_{qc}. Note that the computation results we obtain via this procedure is equivalent to directly working with $\tilde{\mathbf{X}}$.

Randomized PCA. Given the standardized genotype matrix $\tilde{\mathbf{X}}$ (which is never explicitly constructed), the final step of Phase 2 is to securely perform PCA to obtain the top principal components (PCs) of the columns of $\tilde{\mathbf{X}}$ (i.e., the left singular vectors of $\tilde{\mathbf{X}}$). These principal component represent broad genetic patterns in the data that are thought to be indicative of population structure. Since PCA is a complex, iterative algorithm, directly performing PCA on $\tilde{\mathbf{X}}$ is infeasible due to the large input dimensions involved in realistic GWAS scenarios. To illustrate, if we were to naively invoke Eigendecompose (Protocol 31) on the covariance matrix $\tilde{\mathbf{X}}\tilde{\mathbf{X}}^T$ to perform PCA, we would need to QR factorize a $n_{\text{qc}} \times n_{\text{qc}}$ matrix a multiple of n_{qc} times. As a result, communication scales cubically in n_{qc}, which is infeasible. In this work, we instead use an efficient randomized algorithm for matrix factorization based on the techniques of [23].

We give a high-level sketch of the RandomizedPCA (Protocol 32) subroutine that we use. First, we obtain a matrix $\mathbf{Q} \in \mathbb{R}^{m_{\text{pca}} \times (\psi + \alpha)}$ whose orthonormal columns satisfy

$$\tilde{\mathbf{X}}\mathbf{Q}\mathbf{Q}^T \approx \tilde{\mathbf{X}}.$$

Note ψ is the desired number of top principal components, and α is a small oversampling parameter (set to 10 in our experiments) used to increase the stability and accuracy of the algorithm. Also, $\psi + \alpha \ll m_{\text{pca}}$. Such $\mathbf{Q}$ can be obtained by projecting $\tilde{\mathbf{X}}$ onto a random subspace and extracting its orthonormal bases via QR factorization. We additionally apply the power iteration procedure to improve the approximation quality, as described in [23]. More precisely,

$$\mathbf{Q} \in \mathbb{R}^{m_{\text{pca}} \times (\psi+\alpha)} := \text{Orthonormal bases of the row space of } \mathbf{\Pi}\tilde{\mathbf{X}}(\tilde{\mathbf{X}}^T\tilde{\mathbf{X}})^{\rho},$$

where $\mathbf{\Pi} \in \mathbb{R}^{(\psi+\alpha)\times m_{\text{pca}}}$ is a random projection matrix publicly known by all CPs (e.g., CountSketch [24], which is our method of choice), and ρ is the number of power iterations (set to 20 in our experiments). To reduce communication, we derive $\mathbf{\Pi}$ using a PRG, and have each of the computing parties generate it locally (from a globally shared PRG seed).

Next, we perform ψ-truncated singular value decomposition (SVD) on the matrix $\mathbf{Z} := \tilde{\mathbf{X}}\mathbf{Q} \approx \mathbf{U}_\psi\mathbf{\Sigma}_\psi\mathbf{V}_\psi^T$, where $\mathbf{U}_\psi\mathbf{\Sigma}_\psi(\mathbf{Q}\mathbf{V}_\psi)^T$ is the ψ-truncated SVD of $\tilde{\mathbf{X}}\mathbf{Q}\mathbf{Q}^T$, which is approximately $\tilde{\mathbf{X}}$. Thus, $\mathbf{U}_\psi$ approximates the desired left singular vectors of $\tilde{\mathbf{X}}$.

In our adaptation of this algorithm, we take a step further and compute the SVD of $\mathbf{Z}$ by the eigendecomposition of an even smaller $(\psi + \alpha) \times (\psi + \alpha)$ matrix

$$\mathbf{Z}^T\mathbf{Z} = \mathbf{Q}'\mathbf{L}'(\mathbf{Q}')^T$$

with eigenvectors $\mathbf{Q}'$ and a diagonal $\mathbf{L}'$ containing the eigenvalues. Assuming the eigenvalues are sorted, we have

$$\mathbf{Q}'_{:,:\psi} = \mathbf{V}_\psi \quad \text{and} \quad \mathbf{L}'_{:\psi,:\psi} = \mathbf{\Sigma}^2_\psi.$$

Thus, we can recover $\mathbf{U}_\psi$ given the eigendecomposition of $\mathbf{Z}^T\mathbf{Z}$ by computing

$$\mathbf{U}_\psi = \mathbf{Z}\mathbf{Q}'_{:,:\psi}(\mathbf{L}'_{:\psi,:\psi})^{-1/2},$$

which concludes our algorithm.

Overall, our modified algorithm reduces the original problem of factorizing a large $n_{\text{qc}} \times m_{\text{pca}}$ matrix $\tilde{\mathbf{X}}$ to an eigendecomposition of a tiny $(\psi + \alpha) \times (\psi + \alpha)$ matrix $\mathbf{Z}^T\mathbf{Z}$ whose dimensions only slightly exceed the number of top principal components we are interested in. Notably, the size of this subproblem does not depend on n_{qc} or m_{pca}, and in typical GWAS scenarios, we expect $\psi + \alpha \leq 20$.

9.5 Phase 3: Association Tests

The final phase of GWAS is computing the association statistic for each SNP, which intuitively quantifies how informative each SNP is for predicting the phenotype of interest. In this work, we compute the χ^2 statistics of Cochran-Armitage (CA) trend test. We use a generalized version of the CA test described in [22] that additionally corrects for covariates, which in our case, includes those provided in the input (e.g., age) as well as the principal components from Phase 2. Correction is performed by regressing out the covariate features from each genotype and phenotype vectors before computing the test statistic.

Given the secret shares of ψ principal components $\mathbf{U}_\psi \in \mathbb{R}^{n_{\text{qc}}\times\psi}$ and representing ℓ input covariate features as a matrix $\mathbf{C} \in \{0,1\}^{n_{\text{qc}}\times\ell}$, the first step is to find the orthonormal bases of the subspace spanned by both. This is achieved by concatenating the two matrices and invoking QRFactorizeRectangle. We denote the resulting bases of the covariate subspace $\mathbf{Q} \in \mathbb{R}^{n_{\text{qc}}\times(\psi+\ell)}$.

Protocol 32: $[\mathbf{U}_\psi]^{(f)} \leftarrow$RandomizedPCA$([\mathbf{A}]^{(f)}, \psi)$

Require: $\mathbf{A} \in \mathbb{R}^{n \times m}$, number of top principal components (of columns in A) to output ψ, oversampling parameter α, number of power iterations ρ

/* Random projection */
$\mathsf{CP}_0, \mathsf{CP}_1, \mathsf{CP}_2$: Sample a $(\psi + \alpha) \times n$ random sketch matrix $\mathbf{\Pi}$ from $\mathrm{PRG}_{\mathsf{CP}_0,\mathsf{CP}_1,\mathsf{CP}_2}$
$[\mathbf{\Pi A}]^{(f)} \leftarrow \mathbf{\Pi}[A]^{(f)}$

/* Power iteration */
for $i = 1, \ldots, \rho$ **do**
  $[\mathbf{\Pi A}(\mathbf{A}^T\mathbf{A})^{i-1}\mathbf{A}^T]^{(f)} \leftarrow \mathsf{Truncate}([\mathbf{\Pi A}(\mathbf{A}^T\mathbf{A})^{i-1}\mathbf{A}^T]^{(f)}[\mathbf{A}^T]^{(f)}, k+f, f)$
  $[\mathbf{\Pi A}(\mathbf{A}^T\mathbf{A})^{i}]^{(f)} \leftarrow \mathsf{Truncate}([\mathbf{\Pi A}(\mathbf{A}^T\mathbf{A})^{i-1}\mathbf{A}^T]^{(f)}[\mathbf{A}]^{(f)}, k+f, f)$
end for
$[\mathbf{Y}]^{(f)} \leftarrow [\mathbf{\Pi}A(A^TA)^\rho]^{(f)}$

/* Dimensionality reduction */
$[\mathbf{Q}]^{(f)}, [\mathbf{R}]^{(f)} \leftarrow \mathsf{QRFactorizeRectangle}([\mathbf{Y}^T]^{(f)})$
$[\mathbf{Z}]^{(f)} \leftarrow \mathsf{Truncate}([\mathbf{A}]^{(f)}[\mathbf{Q}]^{(f)}, k+f, f)$
$[\mathbf{Z}^T\mathbf{Z}]^{(f)} \leftarrow \mathsf{Truncate}([\mathbf{Z}^T]^{(f)}[\mathbf{Z}]^{(f)}, k+f, f)$

/* Eigendecomposition of reduced matrix */
$[\mathbf{Q}']^{(f)}, [L']^{(f)} \leftarrow \mathsf{Eigendecompose}([\mathbf{Z}^T\mathbf{Z}]^{(f)})$
$[\mathbf{Q}']^{(f)} \leftarrow [\mathbf{Q}'_{:,:\psi}]^{(f)}$ and $[\mathbf{L}']^{(f)} \leftarrow [L'_{:\psi,:\psi}]^{(f)}$

/* Reconstruction of principal components */
$[(\mathbf{L}')^{1/2}]^{(f)}, [(\mathbf{L}')^{-1/2}]^{(f)} \leftarrow \mathsf{SqrtAndSqrtInverse}([L']^{(f)})$ (only on diagonal elements)
$[\mathbf{ZQ}']^{(f)} \leftarrow \mathsf{Truncate}([\mathbf{Z}]^{(f)}[\mathbf{Q}']^{(f)}, k+f, f)$
$[\mathbf{ZQ}'(\mathbf{L}')^{-1/2}]^{(f)} \leftarrow \mathsf{Truncate}([\mathbf{ZQ}']^{(f)}[(\mathbf{L}')^{-1/2}]^{(f)}, k+f, f)$
return $[\mathbf{ZQ}'(\mathbf{L}')^{-1/2}]^{(f)}$

Let $\mathbf{x}_j \in \{0,1,2\}^{n_{\mathrm{qc}}}$ denote the genotype vector (minor allele dosages) of SNP j and $\mathbf{y} \in \{0,1\}^{n_{\mathrm{qc}}}$ be the phenotype vector of interest. Both vectors are first corrected for covariates by projecting them onto the null space of $\mathbf{Q}$:

$$\hat{\mathbf{x}}_j := (\mathbf{I}_{n_{\mathrm{qc}} \times n_{\mathrm{qc}}} - \mathbf{QQ}^T)\mathbf{x}_j,$$
$$\hat{\mathbf{y}} := (\mathbf{I}_{n_{\mathrm{qc}} \times n_{\mathrm{qc}}} - \mathbf{QQ}^T)\mathbf{y}.$$

Next, the CA statistic is computed as the squared Pearson correlation coefficient between the corrected vectors $\hat{\mathbf{x}}_j$ and $\hat{\mathbf{y}}$, which can be expressed as

$$r_j^2 := \frac{\left(n_{\mathrm{qc}} \sum_i \hat{\mathbf{x}}_{ji}\hat{\mathbf{y}}_i - \sum_i \hat{\mathbf{x}}_{ji} \sum_i \hat{\mathbf{y}}_i\right)^2}{\left[n_{\mathrm{qc}} \sum_i \hat{\mathbf{x}}_{ji}^2 - \left(\sum_i \hat{\mathbf{x}}_{ji}\right)^2\right]\left[n_{\mathrm{qc}} \sum_i \hat{\mathbf{y}}_i^2 - \left(\sum_i \hat{\mathbf{y}}_i\right)^2\right]}. \quad (4)$$

We include missing genotypes as zeros to maintain consistency with the PCA step. Note only SNPs and individuals with low missing rates are considered here as a result of Phase 1, so the impact of missing data is minimal.

Efficiently computing the above expression for r_j^2 for every SNP is not trivial. For instance, explicitly constructing the corrected genotypes $\hat{x}_j$ for all SNPs should be avoided, as it would incur

a communication cost equal to the size of the genotype matrix, which is naturally quadratic in the input dimensions. Here, we provide an alternative formulation for computing the CA statistics that requires only linear communication bandwidth.

First, observe that $\hat{\mathbf{y}}$ can be explicitly computed, since it is only a single vector (unlike $\hat{\mathbf{x}}_j$, which exists for each $j \in \{1, \ldots, m_{\text{qc}}\}$). We evaluate $\hat{\mathbf{y}}$ as $\mathbf{y} - \mathbf{Q}(\mathbf{Q}^T\mathbf{y})$ to ensure that all intermediary results have linear size. Once the secret shares of $\hat{\mathbf{y}}$ are obtained, they are Beaver partitioned to use in future computation.

Next, note that computing the following summary statistics are sufficient for computing the CA statistics:

$$\begin{aligned}
s_j^x &:= \textstyle\sum_i \hat{\mathbf{x}}_{ji}, \forall j \in \{1, \ldots, m_{\text{qc}}\}, \\
s_j^{xx} &:= \textstyle\sum_i \hat{\mathbf{x}}_{ji}^2, \forall j \in \{1, \ldots, m_{\text{qc}}\}, \\
s_j^{xy} &:= \textstyle\sum_i \hat{\mathbf{x}}_{ji}\hat{y}_i, \forall j \in \{1, \ldots, m_{\text{qc}}\}, \\
s^y &:= \textstyle\sum_i \hat{\mathbf{y}}_i, \\
s^{yy} &:= \textstyle\sum_i \hat{\mathbf{y}}_i^2.
\end{aligned}$$

We can directly calculate s^y and s^{yy} from the secret shares of $\hat{\mathbf{y}}$. In addition, note s_j^{xy} can be computed using the uncorrected $\mathbf{x}_j$, since taking the inner product with $\hat{\mathbf{y}}$, which is already projected onto the null space of $\mathbf{Q}$, ensures that the component of $\mathbf{x}$ residing in the covariate space will be ignored. Thus, we compute s_j^{xy} as $\sum_i \mathbf{x}_{ji}\hat{\mathbf{y}}_i$ for all j, which is a depth-one circuit with output size m_{qc}.

It remains to show how to compute s_j^x and s_j^{xx}. Let $\mathbf{X}$ be an $n_{\text{qc}} \times m_{\text{qc}}$ matrix constructed by horizontally concatenating $\mathbf{x}_j$ for all j (i.e., $\mathbf{x}_j$ is placed in the j-th column of $\mathbf{X}$). We express the required computation in terms of matrices as

$$\begin{aligned}
\mathbf{s}^x &= \mathbf{1}_{n_{\text{qc}}}^T(\mathbf{I}_{n_{\text{qc}} \times n_{\text{qc}}} - \mathbf{Q}\mathbf{Q}^T)\mathbf{X} \\
&= \mathbf{1}_{n_{\text{qc}}}^T\mathbf{X} - (\mathbf{1}_{n_{\text{qc}}}^T\mathbf{Q}\mathbf{Q}^T)\mathbf{X}, \\
\mathbf{s}^{xx} &= \text{diag}(\mathbf{X}^T(\mathbf{I}_{n_{\text{qc}} \times n_{\text{qc}}} - \mathbf{Q}\mathbf{Q}^T)\mathbf{X}) \\
&= \text{diag}(\mathbf{X}^T\mathbf{X}) - \text{diag}((\mathbf{Q}^T\mathbf{X})^T(\mathbf{Q}^T\mathbf{X})),
\end{aligned}$$

where we write $\mathbf{1}_{n_{\text{qc}}}$ to denote the n_{qc}-dimensional vector consisting of all ones. Next, we simplify the computation by defining

$$\begin{aligned}
\mathbf{u} &:= \mathbf{Q}\mathbf{Q}^T\mathbf{1}_{n_{\text{qc}}}, \\
\mathbf{B} &:= \mathbf{Q}^T\mathbf{X},
\end{aligned}$$

both of which have size linear in the input dimensions and thus, can be evaluated without incurring a quadratic communication cost. The remainder of the computation is computed as

$$\begin{aligned}
\mathbf{s}^x &= \mathbf{1}_{n_{\text{qc}}}^T\mathbf{X} - \mathbf{u}^T\mathbf{X}, \\
\mathbf{s}^{xx} &= \text{diag}(\mathbf{X}^T\mathbf{X}) - \text{diag}(\mathbf{B}^T\mathbf{B}),
\end{aligned}$$

which are depth-one circuits with output size only m_{qc} each.

After all of the summary statistics have been computed, the CA statistics can be obtained via secure multiplications and invocations of SqrtAndSqrtInverse (or Divide) over length-m_{qc} vectors in accordance with Eq. (4). We formally describe our procedure CochranArmitage for efficiently computing CA statistics in Protocol 33.

Protocol 33: $[r^2]^{(f)} \leftarrow$ CochranArmitage($[\mathbf{X}]$, $[\mathbf{y}]$, $[\mathbf{U}_\psi]^{(f)}$, $[\mathbf{C}]$)

Require: Genotype matrix $\mathbf{X} \in \{0,1,2\}^{n \times m}$, phenotype vector $\mathbf{y} \in \{0,1\}^{n \times 1}$, top ψ principal components $\mathbf{U}_\psi \in \mathbb{R}^{n \times \psi}$, additional covariate matrix $\mathbf{C} \in \{0,1\}^{n \times \ell}$

1: /* Obtain covariate subspace */
2: $[\mathbf{U}']^{(f)} \leftarrow \left[[\mathbf{U}]^{(f)} \quad 2^f[\mathbf{C}]\right] \in \mathbb{Z}_q^{n \times (\psi+\ell)}$
3: $[\mathbf{Q}]^{(f)}, [\mathbf{R}]^{(f)} \leftarrow$ QRFactorizeRectangle($[\mathbf{U}']^{(f)}$)
4:
5: /* Calculate corrected phenotypes */
6: $[\mathbf{Q}^T\mathbf{y}]^{(f)} \leftarrow [\mathbf{Q}^T]^{(f)}[\mathbf{y}]$
7: $[\mathbf{Q}\mathbf{Q}^T\mathbf{y}]^{(f)} \leftarrow$ Truncate($[\mathbf{Q}]^{(f)}[\mathbf{Q}^T\mathbf{y}]^{(f)}, k+f, f$)
8: $[\hat{\mathbf{y}}]^{(f)} \leftarrow 2^f[\mathbf{y}] - [\mathbf{Q}\mathbf{Q}^T\mathbf{y}]^{(f)}$
9: $[\mathbf{u}]^{(f)} \leftarrow$ Truncate($[\mathbf{Q}]^{(f)}[\mathbf{Q}^T]^{(f)}\mathbf{1}_{(\psi+\ell)}, k+f, f$)
10:
11: /* Calculate summary statistics */
12: $[\mathbf{s}^x]^{(f)} \leftarrow 2^f\mathbf{1}_{n_{\text{qc}}}^T[\mathbf{X}] - [\mathbf{u}^T]^{(f)}[\mathbf{X}]$
13: $[\mathbf{s}^{xy}]^{(f)} \leftarrow [\hat{\mathbf{y}}^T]^{(f)}[\mathbf{X}]$
14: $[\mathbf{s}^{xx}]^{(f)} \leftarrow \mathbf{1}_n^T([\mathbf{X}] \circ [\mathbf{X}])$, where $[\mathbf{X}] \circ [\mathbf{X}]$ denotes the *element-wise* product of $\mathbf{X}$ with $\mathbf{X}$
15: $[\mathbf{B}]^{(f)} \leftarrow [\mathbf{Q}^T]^{(f)}[\mathbf{X}]$
16: $[\text{diag}(\mathbf{B}^T\mathbf{B})]^{(f)} \leftarrow$ Truncate($\mathbf{1}_{(\psi+\ell)}^T([\mathbf{B}]^{(f)} \circ [\mathbf{B}]^{(f)}), k+f, f$)
17: $[\mathbf{s}^{xx}]^{(f)} \leftarrow [\mathbf{s}^{xx}]^{(f)} - [\text{diag}(\mathbf{B}^T\mathbf{B})]^{(f)}$
18: $[\mathbf{s}^y]^{(f)} \leftarrow [\hat{\mathbf{y}}^T]^{(f)}\mathbf{1}_n$
19: $[\mathbf{s}^{yy}]^{(f)} \leftarrow$ Truncate($[\hat{\mathbf{y}}^T]^{(f)}[\hat{\mathbf{y}}]^{(f)}, k+f, f$)
20:
21: /* Calculate test statistics */
22: $[\bar{\mathbf{s}}^y]^{(f)} \leftarrow$ Truncate($E_f(\frac{1}{N})[\mathbf{s}^y]^{(f)}, k+f, f$)
23: $[\bar{\mathbf{s}}^x]^{(f)} \leftarrow$ Truncate($E_f(\frac{1}{N})[\mathbf{s}^x]^{(f)}, k+f, f$)
24: $[\mathbf{a}]^{(f)} \leftarrow [\mathbf{s}^{xy}]^{(f)} - N \cdot$ Truncate($[\bar{\mathbf{s}}^x]^{(f)}[\bar{\mathbf{s}}^y]^{(f)}, k+f, f$)
25: $[\mathbf{b}_1]^{(f)} \leftarrow [\mathbf{s}^{yy}]^{(f)} - N \cdot$ Truncate($[\bar{\mathbf{s}}^p]^{(f)}[\bar{\mathbf{s}}^y]^{(f)}, k+f, f$)
26: $[\mathbf{b}_2]^{(f)} \leftarrow [\mathbf{s}^{xx}]^{(f)} - N \cdot$ Truncate($[\bar{\mathbf{s}}^x]^{(f)}[\bar{\mathbf{s}}^x]^{(f)}, k+f, f$)
27: $[\sqrt{\mathbf{b}_1}]^{(f)}, [1/\sqrt{\mathbf{b}_1}]^{(f)} \leftarrow$ SqrtAndSqrtInverse($[\mathbf{b}_1]^{(f)}$) (in parallel with Line 28)
28: $[\sqrt{\mathbf{b}_2}]^{(f)}, [1/\sqrt{\mathbf{b}_2}]^{(f)} \leftarrow$ SqrtAndSqrtInverse($[\mathbf{b}_2]^{(f)}$)
29: $[1/\sqrt{\mathbf{b}}]^{(f)} \leftarrow$ Truncate($[1/\sqrt{\mathbf{b}_1}]^{(f)}[1/\sqrt{\mathbf{b}_2}]^{(f)}, k+f, f$)
30: $[\mathbf{r}]^{(f)} \leftarrow$ Truncate($[\mathbf{a}]^{(f)}[1/\sqrt{\mathbf{b}}]^{(f)}, k+f, f$)
31: $[\mathbf{r}^2]^{(f)} \leftarrow$ Truncate($[\mathbf{r}]^{(f)}[\mathbf{r}]^{(f)}, k+f, f$)
32: **return** $[\mathbf{r}^2]^{(f)}$

9.6 Output Reconstruction

At the end of our GWAS protocol, the computing parties CP_1 and CP_2 reveal their individual shares of the association statistics via ReconstructSecret (Protocol 2) and publish the results. In addition, they publish the results of the quality control filters from Phase 1 to facilitate the interpretation of the released GWAS statistics. For instance, one may wish to distinguish whether a particular SNP is deemed insignificant based on the association test or simply excluded from the analysis due to poor quality.

9.7 Precomputation

None of the computation by CP_0 in our overall protocol for GWAS depends on input values, and thus can be performed entirely in advance in a preprocessing phase. The one exception is the filtering step in Phase 1. The auxiliary computing party CP_0 needs the results of the filtering step to obtain the data dimensions used in the subsequent computation and use the appropriate Beaver partitions obtained during the initial data sharing phase. Thus, CP_0 performs the precomputation in stages: once before initial data sharing and once after each filtering stage in Phase 1. Note CP_0 can remain offline for the entirety of Phases 2 and 3.

9.8 Overall Complexity

To summarize, excluding the initial data sharing phase, our secure GWAS protocol requires communication complexity for both precomputation and online computation phases only linear in number of individuals and number of SNPs in the data. This is enabled by the following technical contributions:

- our generalized Beaver partitioning method, which allows for efficient evaluations of depth-one circuits (e.g., matrix multiplication) and effective reuse of Beaver partitioned data for multiple operations (Supplementary Note 3),
- our use of shared random streams (PRGs) to enable Beaver partitioning without any communication between CP_0 and the other computing parties (Supplementary Note 7),
- our use of a randomized PCA algorithm for population stratification analysis, which reduced a large matrix factorization problem to a tiny *constant-sized* matrix (Section 9.4), and
- our careful restructuring of the required computations to ensure that all intermediary results have linear sizes (e.g., Section 9.5).

9.9 Privacy Guarantees

The security of our end-to-end GWAS protocol reduces to the security of the underlying building blocks we use. More precisely, the view of each computing party in each of the subprotocols consist of uniformly random values (or values that are statistically close to uniform). As explained in the security section of Supplementary Note 5, we choose a large enough statistical security parameter κ to achieve an *overall* statistical distance (or equivalently, an adversary's distinguishing advantage) $< 2^{-30}$ between each computing party's view and an ideal distribution where all of the messages exchanged in the protocol consist of uniformly random values. This ensures no information about the underlying input genotypes and phenotypes is leaked to the computing parties during the computation.

Therefore, the only information about the input that is "leaked" are the publicly-revealed GWAS results computed by our protocol, which include the association test statistics and the output of the quality control filters. The quality control results consist of binary-valued inclusion/exclusion status of each individual or SNP, and in realistic GWAS scenarios, do not pose a significant privacy risk for the participants. For example, our per-individual filter reveals only whether a study participant had poor genotyping (too many missing genotypes) or has too many or too few heterozygous sites across the whole genome. The link between such high-level and limited information (a single bit) and the raw genotypes at individual SNPs is extremely tenuous. On the other hand, the association results arguably contain more information about the raw genotypes. We can compose

our protocols with techniques based on differential privacy [25] (as a post-processing step) to assuage these concerns [26, 27]. However, at the scale of a million individuals or more, we expect the risk of releasing such summary statistics to be considerably small.

Nature Biotechnology: doi:10.1038/nbt.4108

Supplementary Note 10: Towards Stronger Security Guarantees

10.1 Relaxing the No-Collusion Assumption

The security of our protocol assumes the computing parties do not collude with each other. In settings where it is difficult to justify this assumption, we can introduce additional computing parties to ensure tolerance against a bounded number of collusions in the online (namely, input-dependent) phase of the computation. Note that we still need to assume a semi-honest (and non-colluding) CP_0 in the precomputation phase. In particular, instead of secret sharing the data between two main computing parties CP_1 and CP_2, we can distribute the private input x across n such entities $\mathsf{CP}_1, \ldots, \mathsf{CP}_n$ such that CP_i for $1 \leq i \leq n-1$ holds an independent and uniformly random number as its share ($[x]_i = r_i$), and CP_n holds $[x]_n = x - \sum_{i=1}^{n-1} r_i$. Analogous to the two-party case, we have the property that $\sum_i [x]_i = x$. Here, as long as there exists a single honest party that does not collude, x remains perfectly hidden. Our building block protocols can be easily extended to handle secret shares over more than two parties. In the extended version of our Beaver partitioning approach, CP_0 still obtains the blinding factors in the clear, so we require that CP_0 does not collude with the other parties. In summary, the relaxed security assumption in the $(n+1)$-party setting (including CP_0) is that CP_0 and at least one other CP are honest (i.e., do not collude with the other parties). The main benefit of this setup is that the protocol is able to tolerate collusion among the other $n-1$ computing parties in the online phase of the computation. As a tradeoff, however, the overall communication scales linearly with the number of computing parties involved.

10.2 Handling Active Adversaries

Our protocol provides security against semi-honest adversaries, namely adversaries that honestly follow the protocol execution, but may subsequently try to learn additional information about other parties' private inputs. In some scenarios, it may be necessary to ensure security even against malicious or active adversaries. For instance, if one of the computing parties is compromised or subverted during the computation, then it may deviate from the protocol specification in order to learn additional information about the participants' genomes.

In the last few years, an elegant line of work [3, 4, 6, 28] has introduced secret-sharing-based multiparty computation protocols with an optimal online phase which provides security against *active* (i.e., malicious) adversaries that may deviate from the protocol execution. The same techniques can be applied to our protocol to achieve security against active adversaries in the online phase of the computation. We describe the main idea used in the SPDZ protocol here [3] and how to extend it to our protocol. For simplicity, we describe our extension in the two-party setting, but everything generalizes naturally to the multiparty setting.

Secret-shared authenticated values. First, in the precomputation phase of the protocol, CP_0 samples a random field element $\alpha \xleftarrow{R} \mathbb{Z}_q$ and secret shares it with the computing parties CP_1 and CP_2. The secret element α serves as a secret key for an information-theoretic message authentication code (MAC) that is used to authenticate the secret-shared data and validate the outputs of the computation. Then, in the online phase of the computation, an authenticated secret-sharing of a value $x \in \mathbb{Z}_q$ is represented as follows:

$$[x] := \langle (\delta, [x]_1, [\alpha(x+\delta)]_1), (\delta, [x]_2, [\alpha(x+\delta)]_2) \rangle \,,$$

Nature Biotechnology: doi:10.1038/nbt.4108

where $\delta \in \mathbb{Z}_q$ is some public scalar (known to all parties). At a high level, each computing party possesses a share of the input x as well as a share of the MAC αx on x.

Computing on authenticated shares. It is straightforward to see that

$$[x] + [y] = [x + y] \quad \text{and} \quad a \cdot [x] = [ax] \quad \text{and} \quad [x] + a = [x + a], \tag{5}$$

where $[x]+[y]$ denotes component-wise addition, $a \cdot [x]$ denotes component-wise scalar multiplication, and

$$[x] + a := \langle (\delta - a, [x]_1 + a, [\alpha(x + \delta)]_1), (\delta - a, [x]_2, [\alpha(x + \delta)]_2) \rangle .$$

Thus, computing linear functions on secret-shared authenticated values is almost identical to computing linear functions of normal secret-shared values (Supplementary Note 2). Multiplication relies on Beaver multiplication triples as before. In particular, given two secret-shared authenticated values $[x]$, $[y]$, and an authenticated sharing of a multiplication triple $([a], [b], [c])$ where $c = ab$, the parties can compute an authenticated secret-sharing of the product $[xy]$. Specifically, the parties first reveal the values $x - a$ and $y - b$ (but *not* their MACs). Then, they compute

$$[xy] := (x - a)(y - b) + (x - a)[b] + (y - b)[a] + [c],$$

exactly as in Protocol 7 using the linear relations defined in Eq. (5). Moreover, our generalization of Beaver multiplication triples (Supplementary Note 3) directly applies to reduce the online communication costs of computing on authenticated values as our generalized Beaver partitioning method only requires computing linear relations over secret-shared authenticated values.

Validating the MAC. At the end of the computation, all of the computing parties have an authenticated secret-sharing $[y]$ of the output y. In the semi-honest version of the protocol, the computing parties would simply publish their shares of the output, and reconstruct the final output of the computation. In the actively-secure protocol, all of the parties first validate the MAC on the output and only if the MAC verification succeeds do the (honest) parties publish their shares. This step of the computation is identical to the output verification step in the SPDZ protocol described in [3, Fig. 1], and we refer the reader to there for the full description. Thus, we can apply the SPDZ techniques to provide active security in the online phase of the computation. The additional computational overhead needed to achieve active security in the online phase of the protocol is essentially a factor of two (since each party has to perform computations on both the secret-shared values as well as their MACs). The communication complexity in the online phase is unchanged since only the blinded inputs (and *not* their MACs) need to be revealed for each Beaver partitioning operation. The protocol also scales naturally to more than two parties (for instance, if we wanted to additionally apply the transformation in the previous section).

Initial data sharing. To leverage the SPDZ techniques for active security in the online setting, we assumed that each of the computing parties have secret-shared *authenticated* values (rather than vanilla secret-shared values). Thus, we need to adapt the initial data sharing procedure (between the study participants SP and the computing parties CP) so that the computing parties possess authenticated shares of the participants' input. We achieve this using a simple adaptation of the SPDZ input-sharing procedure. To simplify the description, assume for now that each SP_i contributes just a single input $x_i \in \mathbb{Z}_q$. The protocol naturally generalizes to the setting where each study participant contributes a vector of field elements. During the precomputation phase, for each study participant SP_i, the auxiliary computing party CP_0 secret-shares a random value $[r_i]$

where $r_i \xleftarrow{R} \mathbb{Z}_q$ with each of the computing parties. To contribute its input to the study, SP_i first interacts with CP_0 to obtain the blinding factor r_i. It then sends the blinded value $x_i - r_i$ to each of the computing parties. Since the computing parties possess an authenticated sharing of $[r_i]$, they can locally compute an authenticated secret-sharing of $[r_i] + (x_i - r_i) = [x_i]$, exactly as required for the online protocol. Note that here, we can use the same trick from Supplementary Note 7 and have CP_0 derive the randomness r_i from a PRG. Then, the total communication between CP_0 and each SP_i consists of only a single (short) PRG seed. Compared to the semi-honest protocol, there is increased communication between CP_0 and one of the computing parties CP_i since CP_0 needs to send over a share of r_i (the other shares can also be derived from a PRG as before). The communication from the study participant to the computing parties also increases, since it now needs to broadcast $x_i - r_i$ to all computing parties. However, the resulting protocol provides stronger security in the online setting.

Security discussion. By relying on the SPDZ protocol for the online phase of the computation, our online protocol is secure against even if one of the computing parties is actively malicious. More generally, in the extended setting with additional computing parties, the online phase provides security against an active adversary that corrupts all but a single computing party. However, the resulting protocol still relies on a semi-honest precomputation phase. In other words, security of the online phase relies on correctly-generated authenticated secret-shared (generalized) Beaver triples, as well as CP_0 not colluding with any of the computing parties in the online phase. While this may seem like a strong assumption, it is important to keep in mind that the precomputation phase is input-independent, and moreover, we only need CP_0 to be a "trusted dealer" (as opposed to a trusted party that possibly sees private inputs). More precisely, CP_0 can be modeled as a "write-only" party since we can structure the protocol such that it *never* needs to receive any message from another party during the protocol execution. This means that an adversary who *only* corrupts CP_0 does not compromise privacy of any of the study participants' inputs.

Supplementary Note 11: Other Cryptographic Frameworks

Here, we provide a ballpark assessment of the applicability of other existing cryptographic frameworks for secure computation, namely homomorphic encryption (HE) [29] and garbled circuits (GC) [12], for securely evaluating large-scale GWAS. HE refers to an encryption scheme that allows certain types of computation to be performed over the private input by manipulating the ciphertexts without decrypting them. Unlike our multiparty solution, HE computation can be carried out by a single party, albeit with greater computational overhead. The current state-of-the-art HE schemes can evaluate a single multiplication in 0.1 seconds [30]. Given that the number of multiplication gates in just the PCA computation is loosely lower-bounded by the number of elements in the genotype matrix, existing HE solutions already require over 30 years of computation to evaluate PCA over a matrix of one million individuals and a reduced set of 10K SNPs, which is clearly infeasible.

On the other hand, Yao's GC protocol is a two-party protocol that enables secure evaluation of arbitrary functionalities (represented as Boolean circuits). In Yao's protocol, one of the parties (i.e., the "garbler") takes the Boolean circuit, and encrypts and permutes (i.e., "garbles") the circuit. The other party (i.e., the "evaluator") is then able to evaluate the garbled circuit and learn the final output, but nothing else about the other party's input. While Yao's protocol require just two rounds of communication for arbitrary computation, the size of the Boolean circuit needed to evaluate our large-scale GWAS computation is prohibitively large. This is due to the fact that GWAS evaluation consists primarily of *arithmetic* operations. Converting an arithmetic circuit to a Boolean circuit incurs a non-negligible overhead, which is typically linear or quadratic in the bit-length of the values. For example, assuming the same number of bits (60) are used to represent a single number as in our method (using the fixed-point representation), a pairwise multiplication using the Karatsuba algorithm requires roughly 600 AND gates. At 128-bits of security, we require 32 bytes to represent a single AND gate in a garbled circuit [31]. Thus, a single pairwise multiplication requires 20.6 KB of communication. Using the same lower-bound for the number of multiplication gates as in our analysis of HE, communicating a garbled circuit for performing PCA requires roughly 190 PB for a million individuals and 10K SNPs. This is well beyond the feasible realm.

Note our estimates above are very loose lower-bounds, and thus we expect the computational burden of the current state-of-the-art HE and GC frameworks for large-scale GWAS to be even greater in practice.

Supplementary Note 12: Towards Logistic Regression Analysis

In case-control studies where the phenotype of interest is binary (e.g., disease status), logistic regression analysis is often used in conjunction with Cochran-Armitage (CA) trend tests to quantify the candidate SNP's impact on phenotypic odds ratio (e.g., disease risk). To this end, one typically trains a logistic regression model for each SNP that predicts the phenotype based on the SNP's minor allele dosage as well as other covariate features, such as age, gender, and population weights (captured by principal components). The estimated model parameter for minor allele dosage is interpreted as the marginal effect of the SNP on the odds ratio and is often reported with top GWAS hits.

Secure evaluation of logistic regression is very challenging. Unlike CA tests, where the required computation can be formulated as a few rounds of matrix multiplications and division (see Protocol 33), training a logistic regression model not only requires frequent evaluation of the sigmoid function (highly nonlinear), but also relies on iterative optimization techniques for parameter estimation (e.g., stochastic gradient descent). These aspects greatly increase the complexity of the overall computation. To illustrate, a recent work based on homomorphic encryption reported a runtime of 30 seconds for a *single* evaluation of the sigmoid function [32]. Applying this technique to GWAS with a million individuals would result in a runtime of almost a year for a *single* pass through the data set, which is clearly not feasible considering that gradient descent methods typically require many passes through the data.

The efficient secure computation techniques we introduced for GWAS offer a more tractable approach for performing logistic regression at large-scale. Notably, we approximate the sigmoid function (more precisely, its logarithm: $-\log(1+\exp(-x))$) as a piecewise-linear function (with 64 segments to ensure high accuracy). Given a private input, we perform secure binary search of depth six—implemented as a sequence of secure comparisons (Protocol 21)—to determine which segment the input belongs to. Then, we retrieve the coefficients of the corresponding linear function via secure table lookup (Protocol 11). Given these coefficients, the (approximate) output of sigmoid and its derivative can be easily computed, non-interactively.

The rest of the computation in logistic regression can be handled by our MPC framework in a straightforward manner. Importantly, our novel generalization of Beaver multiplication triples (Supplementary Note 3) is critical for obtaining an efficient protocol for the stochastic gradient descent (SGD) algorithm, which heavily depends on matrix multiplications and displays high data reuse patterns. In particular, a naïve approach using Beaver multiplication triples would freshly blind/Beaver-partition the input matrix for every iteration, which is infeasible at our scale; with our technique, Beaver-partitioning is performed only once.

Even with our advances, logistic regression imposes an overwhelming computational burden when applied to hundreds of thousands of SNPs in a typical GWAS data set. In practice, we suggest a two-step approach where CA tests are first used to narrow down the set of candidate SNPs with tangible association signals and we consider only the chosen SNPs in a subsequent logistic regression analysis.

We implemented logistic regression in our secure MPC framework and tested it on our benchmark lung cancer data set. Our protocol accurately computed the odds ratios for 100 SNPs within a day of runtime (Supplementary Figure 2). Extrapolating to a million-individual data set, we expect a runtime of approximately three months for computing the odds ratios for 100 SNPs. Note that the actual runtime may be substantial shorter as the number of passes through the data ("epochs") until the convergence of SGD may be smaller for larger data sets given that our models have only few predictive features. While further improvements are needed to achieve genome-wide scalability of secure logistic regression, our results suggest that obtaining the odds ratios for a small subset of

SNPs is currently feasible as newly enabled by our techniques.

Recently, Mohassel and Zhang also introduced an implementation of privacy-preserving logistic regression by combining techniques from secret-sharing-based MPC with garbled circuits [33]. Although they show that their protocol achieves practical runtimes for data sets containing up to a million training instances, their improved scalability comes at the expense of accuracy. In particular, a key factor that contributed to the scalability of their protocol is their use of a coarse approximation of the sigmoid function (namely, as a piecewise linear function with *three* segments). While this approximation may suffice for obtaining competitive *predictive performance* (the focus of their work) for certain data sets, it is too inaccurate in our GWAS setting where the goal is to obtain an accurate estimate of the model parameters. Moreover, even with their coarse approximation, training a separate logistic regression model for each of the hundreds of thousands of SNPs in a typical GWAS data set still requires several years of computation. We also observed that, using our implementation, we can achieve runtimes that are comparable to their approach by reducing the quality of our approximation of the sigmoid function and taking a similar number of passes through the data set for SGD. This further illustrates the challenges of achieving a practical solution for genome-wide logistic regression analysis under secure computation frameworks.

Nature Biotechnology: doi:10.1038/nbt.4108

Supplementary References

[1] Ben-Or, M., Goldwasser, S. & Wigderson, A. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, 1–10 (1988).

[2] Bendlin, R., Damgård, I., Orlandi, C. & Zakarias, S. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, 169–188 (2011).

[3] Damgård, I., Pastro, V., Smart, N. P. & Zakarias, S. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 643–662 (2012).

[4] Damgård, I. *et al.* Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, 1–18 (2013).

[5] Nielsen, J. B., Nordholt, P. S., Orlandi, C. & Burra, S. S. A new approach to practical active-secure two-party computation. In *CRYPTO*, 681–700 (2012).

[6] Keller, M., Orsini, E. & Scholl, P. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS*, 830–842 (2016).

[7] Beaver, D. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 420–432 (1991).

[8] Kamara, S., Mohassel, P. & Raykova, M. Outsourcing multi-party computation. *IACR Cryptology ePrint Archive* 272 (2011).

[9] Bogdanov, D., Laur, S. & Willemson, J. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, 192–206 (2008).

[10] Chandra, A., Fortune, S. & Lipton, R. Lower bounds for constant depth circuits for prefix problems. *Automata, Languages and Programming* 109–117 (1983).

[11] Damgard, I., Fitzi, M., Kiltz, E., Nielsen, J. B. & Toft, T. Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation. In *Theory of Cryptography*, 285–304 (Springer, 2006).

[12] Yao, A. C. Protocols for secure computations (extended abstract). In *FOCS*, 160–164 (1982).

[13] Catrina, O. & Saxena, A. Secure computation with fixed-point numbers. In *International Conference on Financial Cryptography and Data Security*, 35–50 (Springer, 2010).

[14] Goldreich, O. *The Foundations of Cryptography - Volume 1, Basic Techniques* (Cambridge University Press, 2001).

[15] Nishide, T. & Ohta, K. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *International Workshop on Public Key Cryptography*, 343–360 (Springer, 2007).

[16] Markstein, P. Software division and square root using goldschmidts algorithms. In *Proceedings of the 6th Conference on Real Numbers and Computers*, vol. 123, 146–157 (2004).

[17] Dahl, M., Ning, C. & Toft, T. On secure two-party integer division. In *International Conference on Financial Cryptography and Data Security*, 164–178 (2012).

Nature Biotechnology: doi:10.1038/nbt.4108

[18] Seiler, M. C. & Seiler, F. A. Numerical recipes in c: the art of scientific computing. *Risk Analysis* **9**, 415–416 (1989).

[19] Ortega, J. M. & Kaiser, H. F. The llt and qr methods for symmetric tridiagonal matrices. *The Computer Journal* **6**, 99–101 (1963).

[20] Wang, T.-L. Convergence of the tridiagonal qr algorithm. *Linear Algebra and Its Applications* **322**, 1–17 (2001).

[21] Laurie, C. C. *et al.* Quality control and quality assurance in genotypic data for genome-wide association studies. *Genetic Epidemiology* **34**, 591–602 (2010).

[22] Price, A. L. *et al.* Principal components analysis corrects for stratification in genome-wide association studies. *Nature Genetics* **38**, 904–909 (2006).

[23] Halko, N., Martinsson, P.-G. & Tropp, J. A. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review* **53**, 217–288 (2011).

[24] Charikar, M., Chen, K. & Farach-Colton, M. Finding frequent items in data streams. *Theoretical Computer Science* **312**, 3–15 (2004).

[25] Dwork, C. Differential privacy. In *ICALP*, 1–12 (2006).

[26] Simmons, S., Sahinalp, C. & Berger, B. Enabling Privacy-Preserving GWASs in Heterogeneous Human Populations. *Cell Systems* **3**, 54–61 (2016).

[27] Simmons, S. & Berger, B. Realizing Privacy Preserving Genome-Wide Association Studies. *Bioinformatics* **32**, 1293–1300 (2016).

[28] Damgård, I. & Nielsen, J. Scalable and unconditionally secure multiparty computation. *Advances in Cryptology-CRYPTO 2007* 572–590 (2007).

[29] Gentry, C. Fully Homomorphic Encryption Using Ideal Lattices. *STOC* (2009).

[30] Chillotti, I., Gama, N., Georgieva, M. & Izabachene, M. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. *ASIACRYPT* **10031**, 3–33 (2016).

[31] Zahur, S., Rosulek, M. & Evans, D. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*, 220–250 (2015).

[32] Bos, J. W., Lauter, K. & Naehrig, M. Private predictive analysis on encrypted medical data. *Journal of Biomedical Informatics* **50**, 234–243 (2014).

[33] Mohassel, P. & Zhang, Y. Secureml: A system for scalable privacy-preserving machine learning. *Cryptology ePrint Archive* 396 (2017).

Nature Biotechnology: doi:10.1038/nbt.4108