Security Limitations of Using Secret Sharing for Data Outsourcing

Jonathan L. Dautrich, Chinya V. Ravishankar

Security Limitations of Using Secret Sharing for Data Outsourcing

Jonathan L. Dautrich and Chinya V. Ravishankar

University of California, Riverside
dautricj@cs.ucr.edu, ravi@cs.ucr.edu

Abstract. Three recently proposed schemes use secret sharing to support privacy-preserving data outsourcing. Each secret in the database is split into n shares, which are distributed to independent data servers. A trusted client can use any k shares to reconstruct the secret. These schemes claim to offer security even when k or more servers collude, as long as certain information such as the finite field prime is known only to the client. We present a concrete attack that refutes this claim by demonstrating that security is lost in all three schemes when k or more servers collude. Our attack runs on commodity hardware and recovers a 8192-bit prime and all secret values in less than an hour for $k = 8$.

1 Introduction

As cloud computing grows in popularity, huge amounts of data are being outsourced to cloud-based database service providers for storage and query management. However, some customers are unwilling or unable to entrust their raw sensitive data to cloud providers. As a result, privacy-preserving data outsourcing solutions have been developed around an *honest-but-curious* database server model. In this model, the server is trusted to correctly process queries and manage data, but may try to use the data it manages for its own nefarious purposes.

Private outsourcing schemes keep raw data hidden while allowing the server to correctly process queries. Queries can be issued only by a trusted client, who has insufficient resources to manage the database locally. Most such schemes use specialized encryption or complex mechanisms, ranging from order-preserving encryption [1], which has limited security and high efficiency, to oblivious RAM [2], which has provable access pattern indistinguishability but poor performance.

Three recent works [3–5] propose outsourcing schemes based on Shamir's secret sharing algorithm [6] instead of encryption. We refer to these works by their authors' initials, HJ [3], AAEMW [4], and TSWZ [5], and in aggregate as the HAT schemes. The AAEMW scheme was also published in [7]. In secret sharing, each sensitive data element, called a *secret*, is split into n *shares*, which are distributed to n data servers. To recover the secret, the client must combine shares from at least k servers. Secret sharing has perfect information-theoretic security when at most $k - 1$ of the n servers *collude* (exchange shares) [6].

Since secret sharing requires only k multiplications to reconstruct a secret, proponents argue that HAT schemes are faster than encryption based schemes.

Other benefits of the HAT schemes include built-in redundancy, as only k of the n servers are needed, and additive homomorphism, which allows SUM queries to be securely processed by the server and returned to the client as a single value. Each of the HAT schemes makes two security claims:

Claim 1. *The scheme achieves perfect information-theoretic security when at most $k-1$ servers collude.*

Claim 2. *When k or more servers collude, the scheme still achieves adequate security as long as certain information used by the secret sharing algorithm, namely a prime p and a vector $\boldsymbol{X}$, are kept private, known only to the client.*

It is doubtful that the HAT schemes truly fulfill Claim 1, as they sort shares by secret value, which certainly reveals some information about the data [8]. Further, the AAEMW scheme [4] uses correlated coefficients in the secret sharing algorithm instead of random ones, which also contradicts this claim. Nevertheless, for the purposes of this work, we assume that Claim 1 holds.

We are primarily concerned with evaluating Claim 2, which asserts that even k or more colluding servers cannot easily recover secrets. Claim 2 is stated in Sect. 4.5 of [3], Sect. 3 of [4], Sects. 2.2 and 6.1 of [5], and Sect. 3 of [7].

1.1 Our Contribution

Our contribution is to demonstrate that all three HAT schemes [3–5] fail to fulfill Claim 2. We give a practical attack that can reconstruct all secrets in the database when k servers collude, even when p and $\boldsymbol{X}$ are kept private. Our attack assumes that the servers know, or can discover, at least $k+2$ secrets. To limit data storage costs, k is kept small ($k \approx 10$), so discovering $k+2$ secrets is feasible (see Sect. 4.2). All three HAT schemes argue that they fulfill Claim 2, so our result provides a much-needed clarification of their security limitations.

The TSWZ scheme [5] argues that if p is known, secrets could be recovered. However, it provides no attack description, and argues that large primes are prohibitively expensive to recover. Our attack recovers 8192-bit primes in less time than [5] needed to recover 32-bit primes. In fact, we can generally recover large primes in less time than the client takes to generate them (Sect. 5.1).

In Sect. 2 we review Shamir's secret sharing algorithm and how it is used for private outsourcing. In Sect. 3 we give assumptions and details of our attack, and we show how to align shares and discover secrets in Sect. 4. We give experimental runtime results in Sect. 5, and discuss possible attack mitigations in Sect. 6. We discuss related work in Sect. 7 and conclude with Sect. 8.

2 Data Outsourcing Using Secret Sharing

We now review Shamir's secret sharing scheme and show how it is used for private data outsourcing in the HAT schemes [3–5]. We use an employee table with m records as our driving example, where queries are issued over the salary attribute. Each salary $s_1, \ldots, s_m$ is a secret that is shared among the n data servers (see Fig. 1).

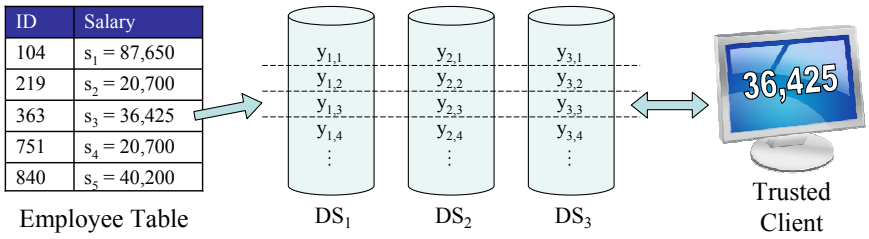

Fig. 1. Secret (salary) data from an employee table is split into shares and distributed to multiple data servers. A trusted client queries shares from the data servers and combines them to recover the secrets.

2.1 Shamir's Secret Sharing

Shamir's secret sharing scheme [6] is designed to share a single secret value s_j among n servers such that shares must be obtained from any k servers in order to reconstruct s_j. The scheme's security rests on the fact that at least k points are needed to uniquely reconstruct a polynomial of degree $k-1$. Theoretically, the points and coefficients used in Shamir's scheme can be taken from any field $\mathbb{F}$. However, to use the scheme on finite-precision machines, we require $\mathbb{F}$ to be the finite field $\mathbb{F}_p$, where p is a prime.

To share s_j, we choose a prime $p > s_j$, and $k-1$ coefficients $a_{1,j}, \ldots, a_{k-1,j}$ selected randomly from $\mathbb{F}_p$. We then construct the following polynomial:

$$q_j(x) = s_j + \sum_{h=1}^{k-1} a_{h,j} x^h \mod p \tag{1}$$

We then generate a vector $\boldsymbol{X} = (x_1, \ldots, x_n)$ of distinct elements in $\mathbb{F}_p$, and for each data server DS_i, we compute the *share* $y_{i,j} = q_j(x_i)$. Together, x_i and $y_{i,j}$ form a point $(x_i, y_{i,j})$ through which polynomial $q_j(x)$ passes.

Given any k such points $(x_1, y_{1,j}), \ldots, (x_k, y_{k,j})$, we can reconstruct the polynomial $q_j(x)$ using Lagrange interpolation:

$$q_j(x) = \sum_{i=1}^{k} y_{i,j} \ell_i(x) \mod p \tag{2}$$

where $\ell_i(x)$ is the Lagrange basis polynomial:

$$\ell_i(x) = \prod_{1 \le j \le k, j \ne i} (x - x_j)(x_i - x_j)^{-1} \mod p \tag{3}$$

and $(x_i - x_j)^{-1}$ is the multiplicative inverse of $(x_i - x_j)$ modulo p.

The secret s_j is the polynomial q_j evaluated at $x = 0$, so we get:

$$s_j = \sum_{i=1}^{k} y_{i,j} \ell_i(0) \mod p \tag{4}$$

Given only $k' < k$ shares, and thus only k' points, we cannot learn anything about s, since for any value of s, we could construct a polynomial of degree $k-1$ that passes through all k' points. Thus Shamir's scheme offers perfect, information-theoretic security against recovering s_j from fewer than k shares [6].

2.2 Data Outsourcing via Secret Sharing

We now describe the mechanism used by all three HAT schemes to support private outsourcing via secret sharing. We first choose a single prime p and a vector $\boldsymbol{X}$, which are the same for all secrets and will be stored locally by the client. For each secret s_j, we generate coefficients $a_{1,j}, \ldots, a_{k-1,j}$, and produce a polynomial $q_j(x)$ as in (1). We then use q_j to split s_j into n shares $y_{1,j}, \ldots, y_{n,j}$, where $y_{i,j} = q_j(x_i)$, and distribute each share $y_{i,j}$ to server DS_i, as in Fig. 1.

An important distinction is that the AAEMW scheme performs secret sharing over the real number field $\mathbb{R}$, so there is no p to choose. However, since the scheme must run on finite precision hardware, any implementation will suffer from roundoff error. Our attack works over $\mathbb{R}$, and is efficient because the field is already known. However, we expect that in practice, the AAEMW scheme will switch to a finite field $\mathbb{F}_p$, so we do not treat it as a special case.

When the client issues a point query for the salary s_j of a particular employee, he receives a share from each of the n servers. Using any k of these shares, he can recover s_j using the interpolation equation (4). Other query types, including range and aggregation queries, are supported by the HAT schemes. We give some relevant details in Sect. 2.4, but the rest can be found in [3–5].

$\boldsymbol{X}$ and p are re-used across secrets for two reasons. First, storing distinct $\boldsymbol{X}$ or p on the client for each secret would require at least as much space as storing the secret itself. Second, when the same $\boldsymbol{X}$ and p are used, the secret sharing scheme has additive homomorphism. That is, if each server DS_i adds shares $y_{i,1} + y_{i,2}$, and we interpolate using those sums, the recovered value is the sum $s_1 + s_2$. With additive homomorphism, when the client issues a SUM query, the server can sum the relevant shares, and return a single value to the client, instead of returning shares separately and having the client perform the addition. Using a different $\boldsymbol{X}$ or p for each secret breaks additive homomorphism.

2.3 Security

If $\boldsymbol{X}$ and p are public, and k or more servers collude, then the HAT schemes are clearly insecure, as the servers could easily perform the interpolation themselves. On the other hand, if at most $k-1$ servers collude, and coefficients are chosen independently at random from $\mathbb{F}_p$, then the servers learn nothing about a secret by examining its shares, and Claim 1 is fulfilled.

The HAT schemes state that by keeping $\boldsymbol{X}$ [3, 4] and p [5] private, they achieve security even when k or more servers collude (Claim 2). Our attack shows that any k colluding servers can recover all secrets in the database, even when $\boldsymbol{X}$ and p are unknown (Sect. 3), contradicting Claim 2.

2.4 Supporting Range and Aggregation Queries

We can use the mechanisms that support range and SUM queries in the HAT schemes to reveal the order of the shares on each server according to their corresponding secret values. We then use these orders to align corresponding shares across colluding servers, and to discover key secret values (see Sect. 4).

The AAEMW scheme [4] crafts coefficients such that the shares preserve the order of their secrets. The HJ and TSWZ schemes [3, 5] both use a single B^+ tree to order each server's shares and facilitate range queries. TSWZ assumes the tree is accessible to all servers, while HJ assumes it is on a separate, non-colluding index server. HJ obscures share order from the servers, but we can reconstruct it by observing range queries over time (see Sect. 4.3).

3 Attack Description

We now show that the HAT schemes are insecure when k or more servers collude, even if $\boldsymbol{X}$ and p are kept private. Our attack efficiently recovers all secret values (salaries in Fig. 1) stored in the database, and relies on the following assumptions:

1. At least k servers collude, exchanging shares or other information.
2. The number of servers k and the number of bits b in prime p are modest: $k \approx 13$, $b \approx 2^{13}$. None of the HAT schemes give recommended values for k or b, with the exception of a brief comment in [4] alluding to 16-bit primes originally suggested by Shamir. In practice, primes with more than 2^{13} bits take longer for the client to generate than for our attack to recover, and the cost of replicating data to every server keeps k small.
3. $\boldsymbol{X}$ and p are unknown, and are the same for each secret (see Sect. 2.2).
4. Each set of k corresponding shares can be *aligned*. That is, the colluding servers know which shares correspond to the same secret, without knowing the secret itself. We can align shares if we know share orders (see Sect. 4.1).
5. At least $k + 2$ secrets, and which shares they correspond to, are known or can be discovered. Since k is modest, knowing $k + 2$ secrets is reasonable, especially when the number of secrets m is large (see Sect. 4.2).

In Sect. 6, we show that modifying the HAT schemes to violate these assumptions sacrifices performance, functionality, or generality, eroding the schemes' slight advantages over encryption based techniques.

3.1 Recovering Secrets when p is Known and X is Private

As a stepping stone to our full attack, we show how to recover secrets if p is already known. Without loss of generality, let $s_1, \ldots, s_k$ be known secrets, and let $DS_1, \ldots, DS_k$ be the colluding servers. For each secret s_j, we have shares $y_{1,j}, \ldots y_{k,j}$, generated by evaluating $q_j(x)$ at $x_1, \ldots, x_k$, respectively. We therefore have a system of k^2 equations of the form $y_{i,j} = s_j + \sum_{h=1}^{k-1} a_{h,j} x_i^h \mod p$, as in (1). The system has $k(k-1)$ unknown coefficients $a_{h,j}$, and k unknown

x_i, giving k^2 equations in k^2 unknowns. Thus, it would seem we can solve for the relevant values of $\boldsymbol{X}$, which would allow us to recover the remaining secrets. Unfortunately, the system is non-linear, so naively solving it directly requires expensive techniques such as Groebner basis computation [9].

Instead, we can recover the remaining secrets without solving for $\boldsymbol{X}$. Consider the following system of equations obtained by applying the interpolation equation (4) to each of the k secrets:

$$\begin{aligned} y_{1,1}\ell_1(0) + y_{2,1}\ell_2(0) + \cdots + y_{k,1}\ell_k(0) - s_1 &\equiv 0 \pmod p \\ y_{1,2}\ell_1(0) + y_{2,2}\ell_2(0) + \cdots + y_{k,2}\ell_k(0) - s_2 &\equiv 0 \pmod p \\ &\vdots \\ y_{1,k}\ell_1(0) + y_{2,k}\ell_2(0) + \cdots + y_{k,k}\ell_k(0) - s_k &\equiv 0 \pmod p \end{aligned} \tag{5}$$

If we treat each basis polynomial value $\ell_i(0)$ as an unknown, we get k unknowns $\ell_1(0), \ldots, \ell_k(0)$, which we call *bases*, in k linear equations. Since we know p, we can easily solve (5) using Gaussian elimination and back-substitution. We can then use the bases to recover the remaining secrets in the database via (4).

We can construct (5) since we know that all shares from a given server DS_i were obtained from the same x_i, and thus should be multiplied by the same base $\ell_i(0)$. The client could obscure the correspondence between shares by mixing shares among servers, but would be forced to store i with each share in order to properly reconstruct the secret. To completely hide the correspondence, i itself would need to be padded and encrypted, which is precisely what secret sharing tries to avoid. Further, mixing the shares would break additive homomorphism.

3.2 Recovering p when $\boldsymbol{X}$ and p are Private

Let b be the number of bits used to represent p. We can easily have $b > 2^6$, so enumerating possible values for p is not practical. However, we can recover p by exploiting known shares and the $k+2$ known secrets. Our attack identifies two composites δ_1 and δ_2 both divisible by p ($p|\delta_1, p|\delta_2$), such that the remaining factors of δ_1, δ_2 are largely independent. We then take δ' to be the greatest common divisor of δ_1 and δ_2, and factor out small primes from δ', leaving us with $\delta' = p$ with high probability. Once p is known, we can use the attack from Sect. 3.1 to recover the bases and the remaining, unknown secrets.

Computing $\boldsymbol{\delta_1, \delta_2}$. Without loss of generality, we let $s_1, \ldots, s_{k+2}$ be the known secrets. To compute δ_γ, $\gamma \in \{1, 2\}$, we consider the system of interpolation equations for secrets $s_\gamma, \ldots, s_{\gamma+k}$ as in (5), represented by the following $(k+1) \times (k+1)$ matrix:

$$\begin{bmatrix} y_{1,\gamma} & y_{2,\gamma} & \cdots & y_{k,\gamma} & -s_\gamma \\ y_{1,\gamma+1} & y_{2,\gamma+1} & \cdots & y_{k,\gamma+1} & -s_{\gamma+1} \\ \vdots & & \ddots & & \vdots \\ y_{1,\gamma+k-1} & y_{2,\gamma+k-1} & \cdots & y_{k,\gamma+k-1} & -s_{\gamma+k-1} \\ y_{1,\gamma+k} & y_{2,\gamma+k} & \cdots & y_{k,\gamma+k} & -s_{\gamma+k} \end{bmatrix} \tag{6}$$

Since p is unknown, we cannot compute inverses modulo p and thus cannot divide as in standard Gaussian elimination. However, we can still convert (6) to upper triangular (row echelon) form using only multiplications and subtractions.

We start by eliminating coefficients for $\ell_1(0)$ from all but the first row ($j = \gamma$). To eliminate $\ell_1(0)$ from row $j > \gamma$, we multiply the contents of row γ through by $y_{1,j}$, and of row j by $y_{1,\gamma}$, producing a common coefficient for $\ell_1(0)$ in both rows. We then subtract the multiplied row γ from the multiplied row j, canceling the coefficient for $\ell_1(0)$. Row 1 is left unchanged, but row j now has coefficient 0 for $\ell_1(0)$, and coefficient $(y_{i,j})(y_{1,\gamma}) - (y_{i,\gamma})(y_{1,j})$ for $\ell_i(0), i \geq 2$:

$$\begin{bmatrix} y_{1,\gamma} & y_{2,\gamma} & \cdots & -s_\gamma \\ 0 & (y_{2,\gamma+1})(y_{1,\gamma}) - (y_{2,\gamma})(y_{1,\gamma+1}) & \cdots & (-s_{\gamma+1})(y_{1,\gamma}) - (-s_\gamma)(y_{1,\gamma+1}) \\ \vdots & & \ddots & \vdots \\ 0 & (y_{2,\gamma+k})(y_{1,\gamma}) - (y_{2,\gamma})(y_{1,\gamma+k}) & \cdots & (-s_{\gamma+k})(y_{1,\gamma}) - (-s_\gamma)(y_{1,\gamma+k}) \end{bmatrix}$$

We then repeat the process, eliminating successive coefficients from lower rows, until the matrix is in upper triangular form:

$$\begin{bmatrix} y_{1,\gamma} & y_{2,\gamma} & \cdots & y_{k,\gamma} & -s_\gamma \\ 0 & c_{2,\gamma+1} & \cdots & c_{k,\gamma+1} & c_{k+1,\gamma+1} \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & c_{k,\gamma+k} & c_{k+1,\gamma+k} \\ 0 & 0 & \cdots & 0 & \delta_\gamma \end{bmatrix} \tag{7}$$

We use $c_{i,j}$ values to denote constants. In the last row of (7), the coefficient for every $\ell_i(0)$ is 0, so the row represents the equation $\delta_\gamma \equiv 0 \pmod{p}$. Thus, $p|\delta_\gamma$.

Size of δ_1, δ_2. As coefficients for successive $\ell_i(0)$ are eliminated, each non-zero cell below the ith row is set to the difference of products of two prior cell values, doubling the number of bits required by the cell. Thus, the number of bits per cell in (7) is given by:

$$\begin{bmatrix} b & b & \cdots & b & b \\ 0 & 2^1 b & \cdots & 2^1 b & 2^1 b \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 2^{k-1} b & 2^{k-1} b \\ 0 & 0 & \cdots & 0 & 2^k b \end{bmatrix}$$

As a result, each of δ_1, δ_2 has at most $2^k b$ bits. This is closely related to the result that in the worst case, simple integer Gaussian elimination leads to entries that are exponential in the matrix size [10].

Recovering p from δ_1, δ_2. Since δ_1, δ_2 have $2^k b$ bits, and p has only b bits, it is likely that δ_1, δ_2 both have some prime factors larger than p, so factoring them directly is not feasible. Instead, we take $\delta' = gcd(\delta_1, \delta_2)$, where gcd is the

greatest common divisor function, which can be computed using the traditional Euclidean algorithm, or more quickly using Stein's algorithm [11].

Since δ_1 and δ_2 were obtained using different elimination orders and sets of secrets, they rarely share large prime factors besides p, so all other prime factors of δ' should be small. Thus, we can factor δ' by explicitly dividing out all prime factors with at most β bits, leaving behind only p, with high probability. We know that p is larger than all shares, so to avoid dividing out p itself, we never divide out primes that are larger than the largest known share. We have found empirically that the probability that δ_1, δ_2, as computed above, share a factor with more than β bits can be approximated by $\frac{2^{(k-2)/4}}{2^{\beta+1}}k$ for the values of β, k we are interested in (Sect. 5.2). Our attack fails if δ_1, δ_2 share such a factor, but we can make the failure rate arbitrarily low by increasing β.

3.3 Attack Complexity

Since δ_1 and δ_2 are both $(2^k b)$-bit integers, the time required to find $gcd(\delta_1, \delta_2)$ is in $O(2^{2k}b^2)$ [11]. As k grows, storing δ_1, δ_2 and computing their gcd quickly become the dominant space and time concerns, respectively. Thus, recovering p has space complexity $O(2^k b)$ and time complexity $O(2^{2k}b^2)$.

Recovering the bases, once p is known, has space complexity $O(k^2 b)$ for storing the matrix, and time complexity dominated either by computing $O(k^3)$ b-bit integer multiplications during elimination, or $O(k)$ modular inverses during back-substitution. Clearly, these costs are dominated by the costs of recovering p. Once p and the bases have been recovered, the time spent recovering a secret is the same for the colluding servers as it is for the trusted client.

3.4 Example Attack for $k = 2$

We now demonstrate our attack on a simple dataset with $m = 6$ records shared over $n = k = 2$ servers. We choose the 6-bit prime $p = 59$ and select $x_1 = 17, x_2 = 39$. We then generate secrets, coefficients, and shares as follows:

$s_1 = 18$	$a_{1,1} = 18$	$q(x_1, s_1) = 29$	$q(x_2, s_1) = 12$
$s_2 = 36$	$a_{1,2} = 5$	$q(x_1, s_2) = 3$	$q(x_2, s_2) = 54$
$s_3 = 22$	$a_{1,3} = 17$	$q(x_1, s_3) = 16$	$q(x_2, s_3) = 36$
$s_4 = 10$	$a_{1,4} = 28$	$q(x_1, s_4) = 14$	$q(x_2, s_4) = 40$
$s_5 = 39$	$a_{1,5} = 31$	$q(x_1, s_5) = 35$	$q(x_2, s_5) = 9$
$s_6 = 57$	$a_{1,6} = 51$	$q(x_1, s_6) = 39$	$q(x_2, s_6) = 40$

We assume s_1, s_2, s_3, s_4 are known. We first generate the matrix in (6) using s_1, s_2, s_3 ($\gamma = 1$), and do the following elimination to get $\delta_1 = 307980$:

$$\begin{bmatrix} 29 & 12 & -18 \\ 3 & 54 & -36 \\ 16 & 36 & -22 \end{bmatrix} \rightarrow \begin{bmatrix} 29 & 12 & -18 \\ 0 & 1530 & -990 \\ 0 & 852 & -350 \end{bmatrix} \rightarrow \begin{bmatrix} 29 & 12 & -18 \\ 0 & 1530 & -990 \\ 0 & 0 & 307980 \end{bmatrix}$$

We do the same with s_2, s_3, s_4 ($\gamma = 2$) to get $\delta_2 = -33984$:

$$\begin{bmatrix} 3 & 54 & -36 \\ 16 & 36 & -22 \\ 14 & 40 & -10 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 54 & -36 \\ 0 & -756 & 510 \\ 0 & -636 & 474 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 54 & -36 \\ 0 & -756 & 510 \\ 0 & 0 & -33984 \end{bmatrix}$$

We then compute $\delta' = gcd(\delta_1, \delta_2) = 2124$, and factor δ' by dividing out the small prime factors $2 \cdot 2 \cdot 3 \cdot 3$, to get $p = 59$, as expected. Now we can recover the bases using the following system of equations as in (5):

$$\begin{aligned} 29\ell_1(0) + 12\ell_2(0) - 18 &\equiv 0 \pmod{59} \\ 3\ell_1(0) + 54\ell_2(0) - 36 &\equiv 0 \pmod{59} \end{aligned}$$

We eliminate $\ell_1(0)$ from the second equation, giving $55\ell_2(0) \equiv 46 \pmod{59}$. We then compute the inverse $(55)^{-1} \pmod{59} = 44$, giving $\ell_2(0) = 46 \cdot 44 \mod 59 = 18$. We then back-substitute to get $\ell_1(0) = 42$. To verify, we compute s_5 and s_6 using (4), giving $s_5 = 35 \cdot 42 + 9 \cdot 18 \pmod{59} = 39$, and $s_6 = 39 \cdot 42 + 40 \cdot 18 \pmod{59} = 57$, as expected.

4 Aligning Shares and Discovering Secrets

In order to mount our attack, we must be able to *align* shares across colluding servers. That is, given the set of shares from each of k servers, we must be able to identify which subsets of k shares, one from each server, were obtained from the same polynomial q_j (1), even if we do not know the secret value s_j itself. Further, we must know, or be able to discover, at least $k+2$ secret values and the subset of k shares to which they correspond. We now show how we can satisfy these assumptions for the HAT schemes [3–5] using knowledge of *share order*.

In the AAEMW [4] and TSWZ [5] schemes, the shares on each server are explicitly, totally ordered (Sect. 2.4). The share order sorts the shares in non-decreasing order of their corresponding secrets. If two shares are obtained from distinct polynomials, but the same secret, they have the same relative order on each server. In the HJ scheme [3], shares are totally ordered, but the order is hidden from the data servers. In this case, we can infer a partial share order by observing queries over time.

4.1 Aligning Shares

When the total share order on each data server is known, we simply align the jth share from each server. If only a partial order is known, as in the HJ scheme, the alignment of some shares will be ambiguous. To recover secrets for such shares, we must either try multiple alignments, or wait for more queries to arrive, and use them to refine the partial order and eliminate the ambiguity (see Section 4.3).

4.2 Discovering $k + 2$ Secrets

We have shown that given $k + 2$ secrets and their corresponding shares, our attack can recover all remaining secrets. This weakness is a severe limitation of the HAT schemes, and contradicts Claim 2 (Sect. 1). In practice, $k \ll m$, where m is the number of secrets, so assuming $k + 2$ known secrets is reasonable. Our attack is independent of the mechanism used to discover these secrets.

Simple methods for learning $k + 2$ secrets include a known plaintext attack, where we convince the trusted client to insert $k + 2$ known secrets, and a known ciphertext attack, where the client reveals at least $k+2$ secrets retrieved by some small range query. Since shares are ordered according to their secret values, we can easily identify which subsets of shares from the query go with each secret.

We can also infer secret values using share order. Consider an employee table with secret salaries, as in Fig. 1. If at least $k + 2$ employees earn a well-known minimum-wage salary, then the share order reveals that the first $k+2$ shares have this known salary. Alternatively, there may be $k+2$ employees who anonymously post their salaries. If we can estimate the distribution of salaries in the database, we can guess roughly where the known salaries fall in the order, and run the attack for nearby guesses until we get a solution with a recoverable prime and recovered secrets that fit the expected order.

4.3 Inferring Order in the HJ Scheme

If a scheme hides the share order from the data server, share alignment and secret discovery become harder. The HJ scheme [3] stores the share order for each data server on a single index server that ostensibly does not collude with any data servers. The client sends each query to the index server, and the response tells the client which shares to request from each data server.

In the simplest case, we can align shares by observing point queries, which return only one share from each server. If the colluding servers all observe an isolated request for a single share at the same time, they can assume the shares satisfy a point query, and thus that they all correspond to the same secret. Given enough point queries, we can align enough shares to mount our attack. However, if point queries are rare, this technique will take too long to be useful.

More generally, we can order shares on each server by observing overlapping range queries. In the HJ scheme, a range query appears to the data server as set of unordered share requests. Since range queries request shares that have secrets inside a given range, we know that secrets of requested shares are contiguous. We use this information to order shares according to their secret values.

Consider an example where a client issues two range queries to the same data server. The first query returns shares $\{y_1, y_2\}$, and the second, shares $\{y_3, y_2\}$. Each query is a range query, so the server knows that no secret can fall between the secrets of y_1 and y_2 or of y_3 and y_2. Since y_2 appears in both queries, the server knows that the secret of y_2 comes between the secrets of y_1 and y_3, but is not sure whether the secret of y_1 or of y_3 is smaller. Thus, the true share order

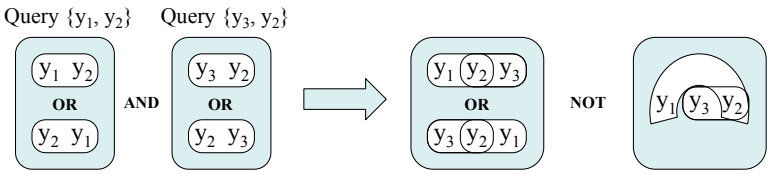

Fig. 2. Range queries indicate that the secrets of shares y_1, y_2 are contiguous, as are those of y_3, y_2. Thus, the secret of y_2 falls between the secrets of y_1 and y_3, though either y_1 or y_3 may have the smallest secret.

contains either subsequence $y_1y_2y_3$ or $y_3y_2y_1$, and we say that the server knows the share order of $\{y_1, y_2, y_3\}$ up to symmetry (see Fig. 2).

We can extend this technique to additional range queries of varying sizes. Given enough queries, we can reconstruct the entire share order on each server up to symmetry. The full reconstruction algorithm uses PQ-trees [12], but is out of scope for this paper. We can link reconstructed share orders across servers, and thereby align shares, by observing that if a query issued to one data server requests the jth share, then the same query must also request the jth share from every other server. If we use the share order to discover secrets, we must make twice as many guesses, since we still only know the order up to symmetry.

5 Attack Implementation and Experiments

We implemented our attack in Java, and ran each of our attack trials using a single thread on a 2.4GHz Intel® Core™2 Quad CPU. All trials used less than 2GB RAM. We used two datasets. The first consists of $m = 1739$ maximum salaries of Riverside County (California, USA) government employees as of February, 2012 [13]. The second is a set of $m = 10^5$ salaries generated uniformly at random from the integer range $[0, 10^7)$.

5.1 Time Measurements

Our first set of experiments measures the time required to run the full attack as described in Sect. 3. Each experiment varies the number of servers k or the number of bits b in the hidden prime p. The total number of servers n has no effect on the attack runtime, so we let $n = k$. All times are averaged over 10 independent trials, and averages are rounded up to a 1ms minimum. In each trial, we divide out primes with at most $\beta = 16$ bits (Sect. 3.2), and we successfully recover p, all k bases ($\ell_i(0)$ values), and all m secrets.

Each plot gives the times spent by the client finding a random b-bit prime p and creating k shares for each of the m secrets. We then plot the times spent by the colluding servers recovering p and the k bases. We also give the time spent recovering all m secrets, which is the same for the colluding servers as it is for the client. From Sect. 3.3, we know that the time needed to recover p is

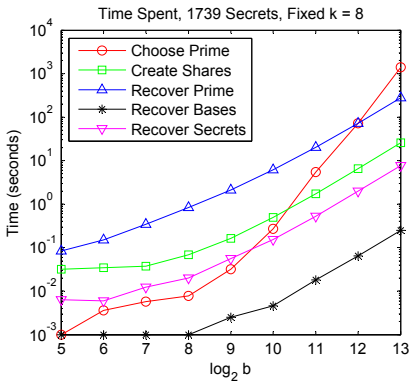

Fig. 3. Riverside dataset times, varied b

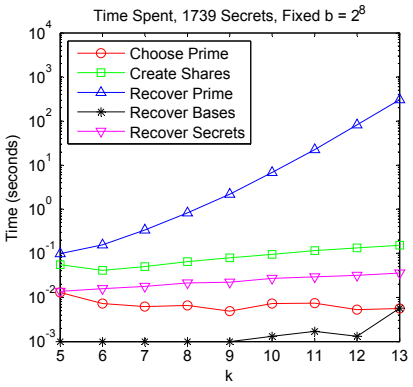

Fig. 4. Riverside dataset times, varied k

in $O(2^{2k}b^2)$. Thus, incrementing k or $\log_2 b$ increases prime recovery time by a factor of 4. Since k and $\log_2 b$ have similar effects on prime recovery time, we plot against $\log_2 b$ instead of b on the x axis.

Figures 3 and 4 plot times using the Riverside dataset with fixed $b = 2^8$ and $k = 8$, respectively. Figures 5 and 6 give corresponding times for the random dataset. Times to create shares and recover secrets are proportional to m, and so are higher for the larger, random dataset. Times to generate p, recover p, and recover bases depend only on b and k, and so are dataset-independent.

Figures 3 and 5 show that when k is held constant, increasing b costs the client more than it costs the colluding servers. Both prime recovery and modular multiplication take time proportional to b^2, so prime recovery time is a constant factor of share generation time. Further, the time to choose a random b-bit prime

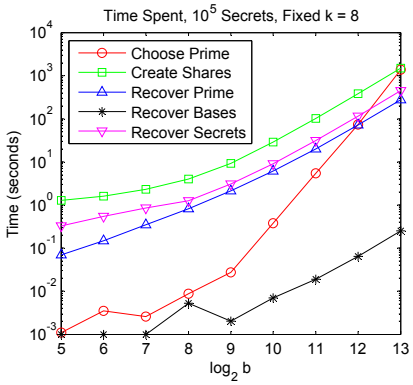

Fig. 5. Synthetic dataset times, varied b

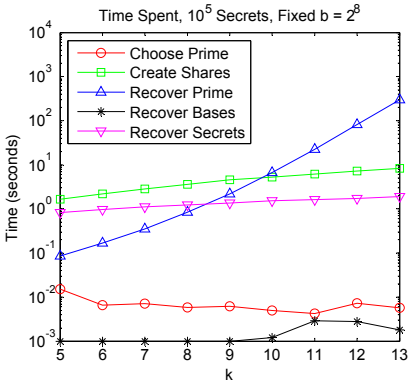

Fig. 6. Synthetic dataset times, varied k

using the Miller-Rabin primality test is in $O(b^3)$ [14], so as b grows past 2^{12}, the cost to generate p quickly outstrips the cost to recover it. Thus it is entirely impractical to thwart our attack by increasing b.

In the TSWZ scheme [5], the measured time to recover a prime with less than 2^5 bits was over 1500 seconds. In contrast, our method recovers primes with 2^{13} bits in under 500 seconds on comparable hardware, for $k = 8$. As long as $k \ll b$, as is likely in practice, our method is far faster.

Figures 4 and 6 show that when b is fixed, most times are in $O(k)$, with the exception of prime recovery time, which is in $O(2^{2k})$. Thus, by increasing k, the attack can be made arbitrarily expensive at a relatively small cost to the client. However, as we discuss in Sect. 6, even $k = 10$ may be impractical.

5.2 Failure Rate Measurements

Since we only factor out small primes with at most β bits (Sect. 3.2), our attack fails if δ_1, δ_2 share any prime factor, other than p, that has more than β bits. Thus, our attack's failure rate r_f is the probability that $\delta_1/p, \delta_2/p$ share a prime with more than β bits. Since δ_1, δ_2 are not independent random numbers, it is difficult to compute r_f analytically, so we measure it empirically. The results of our experiment are shown in Fig. 7.

We found that r_f is largely independent of b, but depends heavily on k and β. To measure r_f, we conducted several trials in which we generated a prime p of $b = 32$ bits, and ran our attack using $k + 2$ randomly generated secrets. For $k = 2$, we ran 10^6 trials, and were able to get meaningful failure rates up through $\beta \approx 16$. Trials with larger k were much more expensive, so we only ran 10^3 trials for $k = 6$ and $k = 10$, and the results are accurate only through $\beta \approx 10$.

From our results, we derived the approximate expression $r_f \approx \frac{2^{(k-2)/4}}{2^{\beta+1}}k$. We then plotted this estimated r_f in Fig. 7, denoted by *est*. The approximation is adequate for the range of β we're considering. The dependence of r_f on $2^{-(\beta+1)}$ is expected, as the probability that a factor of $\beta+1$ bits found in one random d-bit

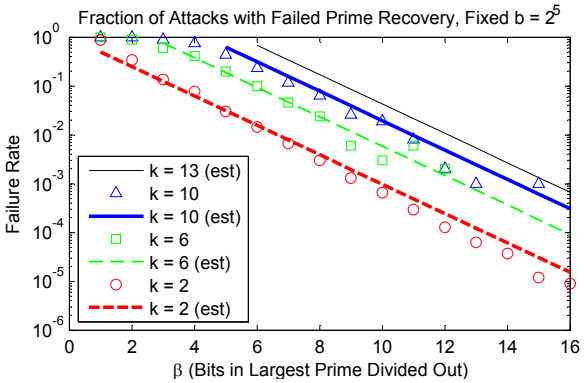

Fig. 7. Attack failure rates for varied k and β

number is found in another random d-bit number is roughly $\frac{2^{d-(\beta+1)}}{2^d} = 2^{-(\beta+1)}$. The nature of the dependence on k is unclear, but it may be related to the fact that k of the $k+1$ equations used to compute δ_1 are also used to compute δ_2.

Using our approximation for r_f, we estimate the worst-case failure rate for our timing experiments, where $\beta = 16$ and $k = 13$, to be $r_f \approx \frac{2^{(13-2)/4}}{2^{16+1}} 13 = 2^{-14.25} 13 \approx 6.67 \times 10^{-4}$. If necessary, we can lower r_f further by increasing β.

6 Attack Mitigations

We now discuss possible modifications a client can make to the HAT schemes that may improve security. In order to mitigate our attack, a modification must cause at least one of the attack assumptions listed in Sect. 3 to be violated.

Assumption: At Least k Servers Collude. The simplest way to thwart our attack is to ensure that no more than $k-1$ servers are able to collude. Only in such cases can secret sharing schemes hope to achieve perfect, information-theoretic security. However, if the number of colluding servers must be limited, secret sharing schemes cannot be applied to the honest-but-curious server threat model commonly used for data outsourcing [1, 2, 8, 15, 16].

Assumption: b, k Modest. In Sect. 5, we showed that increasing b costs the client more than it costs the colluding servers, so a large b is impractical. With limited resources, we successfully mounted attacks for $k = 13$ in under 500 seconds, so k must be substantially larger ($k > 20$) to achieve security in practice. For each server, the client pays a storage cost equal to that of storing his data in plaintext. If $k \geq 10$, the combined storage cost exceeds that of many encryption-based private query techniques [1, 2, 15], so increasing k is also impractical.

Assumption: Same $\boldsymbol{X}, p$ for Each Secret. Storing a distinct $\boldsymbol{X}$ or prime p on the client for each secret is at least as expensive as storing the secret itself. An alternative is to use a strong, keyed hash h_j to generate a distinct vector $\boldsymbol{X}' = h_j(\boldsymbol{X})$ for each secret s_j. Using this method, each secret requires different basis polynomials for interpolation, so mounting an attack would be much harder. Unfortunately, it also eliminates additive homomorphism, removing support for server-side aggregation, which is cited as a reason for adopting secret sharing.

Assumption: Corresponding Shares can be Aligned. Hiding share order from data servers as in [3] can hinder share alignment, but if the scheme supports range or point queries, share alignment can eventually be inferred (Sect. 4.3). Schemes could use re-encryption or shuffling to obscure order as in [2], but the cost of such techniques outweighs the performance advantages of secret sharing.

Assumption: $k+2$ Known Secrets. It is difficult to keep all secrets hidden from an attacker. Known plaintext/ciphertext attacks for small amounts of data are always a threat, and if we known the real-world distribution of the secrets, we can guess them efficiently (Sect. 4.2). The client could encrypt secrets before sharing, but doing so adds substantial cost and eliminates additive homomorphism.

7 Related Work

Privacy-preserving data outsourcing was first formalized in [16] with the introduction of the *Database As a Service* model. Since then, many techniques have been proposed to support private querying [1, 2, 15, 17, 18], most based on specialized encryption techniques. For example, order-preserving encryption [1] supports efficient range queries, while [15] supports server-side aggregation through additively homomorphic encryption. Other schemes are based on fragmentation, where only links between sensitive and identifying data are encrypted [17, 18].

As far as we know, the schemes we discuss in this paper [3–5, 7] are the first to use secret sharing to support private data outsourcing, though secret sharing has been used for related problems, such as cooperative query processing [19]. Prior works, such as [8], have addressed various security issues surrounding data outsourcing schemes, but as far as we know, ours is the first to reveal the specific limitations of schemes based on secret sharing.

8 Conclusion

Private data outsourcing schemes based on secret sharing have been advocated because of their slight advantages over existing encryption-based schemes. Such advantages include security, speed, and support for server-side aggregation. All three outsourcing schemes based on secret sharing [3–5] claim that security is maintained even when k or more servers collude. To the contrary, we have shown that all three schemes are highly insecure when k or more servers collude, regardless of whether $\boldsymbol{X}$ and p are kept secret.

We described and implemented an attack that reconstructs all secret data when only $k + 2$ secrets are known initially. In less than 500 seconds, our attack recovers a hidden 256-bit prime for $k \leq 13$ servers, or an 8192-bit prime for $k \leq 8$. We discussed possible modifications to mitigate our attack and improve security, but any such modifications sacrifice generality, performance, or functionality.

We conclude that secret sharing outsourcing schemes are not simultaneously secure and practical in the honest-but-curious server model, where servers are not trusted to keep data private. Such schemes should only be used when the client is absolutely confident that at most $k - 1$ servers can collude.

Acknowledgements. This work was supported in part by contract number N00014-07-C-0311 with the Office of Naval Research.

References

1. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In: Proc. ACM SIGMOD. (2004) 563–574
2. Stefanov, E., Shi, E., Song, D.: Towards practical oblivious RAM. Proc. NDSS (2012)

3. Hadavi, M., Jalili, R.: Secure Data Outsourcing Based on Threshold Secret Sharing; Towards a More Practical Solution. In: Proc. VLDB PhD Workshop. (2010) 54–59
4. Agrawal, D., Abbadi, A., Emekci, F., Metwally, A., Wang, S.: Secure data management service on cloud computing infrastructures. New Frontiers in Information and Software as Services (2011) 57–80
5. Tian, X., Sha, C., Wang, X., Zhou, A.: Privacy preserving query processing on secret share based data storage. In: Proc. DASFAA, Springer (2011) 108–122
6. Shamir, A.: How to share a secret. Communications of the ACM (1979) 612–613
7. Agrawal, D., El Abbadi, A., Emekci, F., Metwally, A.: Database Management as a Service: Challenges and Opportunities. In: Proc. ICDE Workshop on Information and Software as Services. (2009) 1709–1716
8. Kantarcioglu, M., Clifton, C.: Security issues in querying encrypted data. Data and Applications Security XIX (2005) 924–924
9. Buchberger, B., Winkler, F.: Gröbner bases and applications. Cambridge University Press (1998)
10. Fang, X., Havas, G.: On the worst-case complexity of integer gaussian elimination. In: Proceedings of the 1997 international symposium on symbolic and algebraic computation, ACM (1997) 28–31
11. Stein, J.: Computational problems associated with racah algebra. Journal of Computational Physics **1**(3) (1967) 397–405
12. Booth, K.S., Lueker, G.S.: Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. J. Comput. System Sci. **13**(3) (1976) 335–379
13. County of riverside class and salary listing. `http://www.rc-hr.com/HRDivisions/Classification/tabid/200/ItemId/2628/Default.aspx` (February 2012)
14. Rabin, M.: Probabilistic algorithm for testing primality. Journal of number theory **12**(1) (1980) 128–138
15. Mykletun, E., Tsudik, G.: Aggregation queries in the database-as-a-service model. In: Proc. DBSec. (2006) 89–103
16. Hacigümüş, H., Iyer, B., Li, C., Mehrotra, S.: Executing SQL over encrypted data in the database-service-provider model. In: Proc. ACM SIGMOD. (2002) 216–227
17. Ciriani, V., De Capitani Di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Keep a few: Outsourcing data while maintaining confidentiality. Proc. ESORICS (2010) 440–455
18. Nergiz, A., Clifton, C.: Query processing in private data outsourcing using anonymization. In: Proc. DBSec. (2011) 138–153
19. Emekci, F., Agrawal, D., Abbadi, A., Gulbeden, A.: Privacy preserving query processing using third parties. In: Proc. ICDE, IEEE (2006) 27–27