Securely Sampling Biased Coins with Applications to Differential Privacy

Jeffrey Champion
Northeastern University
champion.j@husky.neu.edu

abhi shelat
Northeastern University
abhi@neu.edu

Jonathan Ullman
Northeastern University
jullman@ccs.neu.edu

ABSTRACT

We design an efficient method for sampling a large batch of d independent coins with a given bias $p \in [0, 1]$. The folklore secure computation method for doing so requires $O(\lambda + \log d)$ communication and computation per coin to achieve total statistical difference $2^{-\lambda}$. We present an exponential improvement over the folklore method that uses just $O(\log(\lambda + \log d))$ gates per coin when sampling d coins with total statistical difference $2^{-\lambda}$. We present a variant of our work that also concretely beats the folklore method for $\lambda \geq 60$ which are parameters that are often used in practice. Our new technique relies on using specially designed oblivious data structures to achieve biased coin samples that take an expected 2 random bits to sample.

Using our new sampling technique, we present an implementation of the differentially private report-noisy-max mechanism [4] (a more practical implementation of the celebrated exponential mechanism [28]) as a secure multi-party computation. Our benchmarks show that one can run this mechanism on a domain of size $d = 2^{12}$ in 6 seconds and up to $d = 2^{19}$ in 14 minutes. As far as we know, this is the first complete distributed implementation of either of these mechanisms.

CCS CONCEPTS

• **Security and privacy** → **Privacy-preserving protocols**; • **Mathematics of computing** → *Probabilistic algorithms.*

KEYWORDS

differential privacy, multi-party computation

ACM Reference Format:
Jeffrey Champion, abhi shelat, and Jonathan Ullman. 2019. Securely Sampling Biased Coins with Applications to Differential Privacy. In *2019 ACM SIGSAC Conference on Computer& Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3319535.3354256

1 INTRODUCTION

This paper presents asymptotically and concretely superior secure computation methods for sampling a batch of d coins with bias $p \in [0, 1]$. The problem of sampling biased coins plays a fundamental role in implementing many randomized algorithms, in running Monte Carlo simulations, and in producing differentially private data summaries. Furthermore, as explained in [14], the tasks of sampling from binomial, Poisson, Laplace, or geometric distributions can all be reduced to the task of sampling biased coins. Thus, we consider this task an essential primitive in the area of secure computation.

Biased Sampling with (and without) Secure Computation. In order to explain our contributions, we provide some background on methods for sampling biased coins and their complexity. If we are willing to sample "in the clear," then the folklore method for sampling a coin with bias p is to first sample a uniform number $r \in [0, 1]$ and then output heads if $r \leq p$ and tails otherwise. A naïve implementation of this first method that achieves error at most $2^{-\lambda}$ is to sample r discretely in the following way: flip λ fair coins, interpret these coins as the binary expansion of r, and perform the comparison. A downside of this method is that it flips λ coins to sample one biased coin, and thus the running time is at least $\Omega(\lambda)$. One can address this issue using the following *lazy flipping* strategy: flip the first coin and compare against the first digit in the binary expansion of p, if the digits differ than a decision can be made, and otherwise, the process is repeated with a fresh coin and the next digit of the expansion. A simple calculation shows that lazy flipping requires just 2 coins and $O(1)$ time in expectation, regardless of λ! Thus, sampling biased coins "in-the-clear" is essentially solved in a Turing-machine model of computation. In this work we focus on achieving complexity similar to the lazy flipping method but in a secure computation.

Unfortunately, the lazy flipping method cannot be easily implemented in a two- or multi-party secure computation. Recall that the goal of Secure Multi-Party Computation is to allow a set of parties $P_1, \ldots, P_n$ to securely evaluate a function $y = f(x_1, \ldots, x_n)$, where each P_i provides input x_i. Here securely evaluating a function means computing the function jointly in such a way that no P_i learns anything other than what is revealed by the output y and their own input x_i. In particular, each P_i must not learn x_j for $i \neq j$, nor any intermediate value derived from x_j during the computation of f. To achieve this property, implementing an MPC version of an algorithm usually requires that all parts of the algorithm be converted into static circuits. This means that loops for example cannot stop early, since the stopping time may reveal something about the inputs.

Since the lazy flipping method implicitly leaks the number of fair coins that were flipped, it cannot be directly implemented as a secure computation while preserving its efficiency. Specifically, if we transform the algorithm to a binary circuit, the circuit that samples r must have size proportional to the worst case where all λ bits are compared against the binary expansion of p. Thus a secure computation for the folklore sampling mechanisms generally requires λ fair coins per sample instead of an expected 2. When expressed in terms of boolean gates—which generically represents the running time and communication complexity of such a protocol—this requirement leads to $O(\lambda)$ gates per coin.

A natural approach to overcome this inefficiency is to use a secure computation protocol along with an oblivious RAM data structure, implemented as a circuit, to emulate lazy sampling. Oblivious RAM data structures, first introduced by Goldreich and Ostrovsky [19] allow the implementation of a RAM program while hiding the addresses of the memory locations that are accessed during the execution. By hiding the memory access pattern, this approach could allow the lazy sampling of coins that requires only an expected 2 fair coins per sample. However, to implement one read operation on a memory of size λ, most ORAM data-structures require polylog(λ) additional read and write memory operations [19, 26, 35, 42].

The state of the art in relation to asymptotic complexity is Panorama [31], which requires $\tilde{O}(\log \lambda)$ extra operations (albeit with astronomically high constants). However, even ignoring the large constants, all known ORAM schemes require read operations on machine words of size $\log(\lambda)$ bits when accessing a memory with λ elements. Since our application only requires reads of single bits to implement lazy sampling, all of the schemes will incur $\tilde{O}(\log^2 \lambda)$ overhead which is asymptotically worse than our scheme. Ignoring asymptotics, the most recent practical implementations of ORAM for secure computation [12] concretely perform worse than our approach (and also only support two-party secure computation).

Our Contributions. The first main contribution of this paper is to develop a new secure computation sampling procedure that takes a string of fair coins[1] and samples a string of d biased coins that has statistical distance at most $2^{-\lambda}$ from a string of d independent coins with bias p, using an *amortized* $O(\log(\lambda + \log d))$ AND gates per coin. This result improves *exponentially* over the folklore technique that uses $O(\lambda + \log d)$ gates per coin, and is polynomially better than schemes that use the state-of-the-art ORAM techniques.

Our main technique is to employ new oblivious data structures that allow us to amortize the cost of the lazy sampling method by "blurring" when the sampling of one biased coin ends the the next begins. In §2.1, we describe these two data structures that enable our improvements: an *oblivious push-only stack*, and an *oblivious pop-only stack with reset.* The first push-only stack allows obliviously pushing elements onto a stack. When the stack is full, the data structure obliviously ignores the operations and does not change the underlying data. The second data structure allows the opposite—it only supports pop operations, and returns the last element repeatedly when it is empty. It additionally supports an *oblivious reset* which returns the stack to an arbitrary original configuration. Both of these data structures are inspired by the oblivious stack proposed by Zahur and Evans [44].

Using these data structures, our new sampling method works as follows. We first initialize a pop-only stack with the binary expansion of the bias p. At each step, we pop from the stack and compare against the next fair coin. We obliviously compare the coins and if they agree, we make an "empty" oblivious push to the push-only stack, and we simply repeat the procedure by popping and comparing with the next fresh fair coin. If the two coins disagree, then we can output a biased coin sample by performing a true oblivious push to the push-only stack and obliviously resetting the pop-only stack. Thus, each iteration of the loop requires one oblivious push, one oblivious pop, one oblivious reset, and one comparison. As we show in §2.1, these operations can all be done in $O(\log(\lambda + \log d))$ gates per coin.

While our method asymptotically beats the standard secure sampling methods, our implementation efforts[2] reported in §5 show that the constant overheads in our careful stack implementation only beat the standard methods when the statistical parameter $\lambda > 200$, i.e., when the sampling error is set to be quite small. As a second contribution, we show that for larger statistical errors $\lambda \in [60, 512]$, an alternative method often beats the naïve strategy. In particular, let $C_p(\cdot)$ be a circuit that on input j produces the j^{th} bit in the binary expansion of bias p. We can replace the pop-only stack that holds the bits of p used in the method described above by this circuit. We show that for an appropriate range of $\lambda \in [60, 512]$, it is indeed possible to build such circuits for any arbitrary bias p. In §5, we utilize circuits that output the first 128 bits of any p using at most 13 AND gates. Such a circuit is simply a 7-bit boolean predicate, and Peralta *et al.* [8] show that every 6-bit predicate can be computed in at most 6 AND gates. Our result follows from simply muxing the top and bottom halves of the 7-bit predicate truth table. In practice, we implemented this for many p and found that all of them could in fact be expressed in 11 gates. However, as the statistical parameter increases, the size of our predicate also increases linearly, and thus this method eventually becomes more expensive than the pop-only data structure. We evaluate the cross-over point and determine this to be $\lambda > 512$. In all cases, these methods surpass the naïve sampling circuit at $\lambda \geq 60$. We summarize all (asymptotic) amortized gate complexities and random coins used to make a single biased coin in Table 1.

Application to Differential Privacy. As an application of our sampling methods, we give improved secure multiparty implementations of fundamental algorithms from *differential privacy* [15]. Differential privacy is a strong formal model of data privacy tailored to statistical applications. Intuitively, a randomized algorithm is differentially private if it does not reveal "too much" about the data of any one individual. At a high level, these algorithms introduce random noise that masks the contribution of one individual, while preserving the overall utility of the dataset when the number of users is sufficiently large. Differential privacy has been the subject of an enormous body of literature (see e.g. [16] for a textbook treatment) and has now been implemented by companies such as Apple [37, 38] and Google [5, 18] and statistical agencies such as the U.S. Census Bureau [22].

The most powerful differentially private algorithms are designed in a *centralized model* where a trusted party collects the data and agrees to publish only the output of the algorithm. In many industrial applications, this trust assumption is problematic, and so companies have mostly opted to use the *local model* [15, 25, 43], which

[1]The fair coins can be obtained securely by taking the XOR of fair coins obtained from each party.

[2]Our code can be found at https://www.gitlab.com/neucrypt/securely_sampling.

is essentially a weak model of information-theoretic secure computation where each party applies a separate differentially private algorithm to their own data. Unfortunately the local model severely limits the utility of the algorithm both in theory [10, 13, 25, 39] and in practice, often requiring billions of users to achieve reasonable utility (see e.g. [5]).

To resolve this tension between the central and local models, the prescient work of Dwork *et al.* [14] posed the question of secure multi-party implementations of differentially private algorithms, and gave algorithms for sampling the noise required to implement simple counting mechanisms. Using our secure sampling methods, we give improved algorithms for sampling the noise in fundamental differentially private algorithms.

In particular, as far as we are aware, we give the first full secure implementation of the *report-noisy-max mechanism* (which is a more practical implementation of the celebrated *exponential mechanism* [28]). This is a highly versatile mechanism that is the driving force in numerous applications of differential privacy (see e.g. [4, 6, 23, 36] for a tiny sample). This mechanism is particularly crucial in applications of distributed differential privacy, as any implementation of this mechanism in the local model provably suffers an exponential loss of utility [25, 39], even in some of its simplest applications. This application is well suited to our methods due to its need for many biased coins and because the need to securely take a maximum makes it more amendable to circuit-based protocols rather than the sorts of tailored algebraic protocols that have been applied to computing sums (e.g. [7, 33, 34]).

Our experiments reported in §5 show that datasets of size 2^{12} up to 2^{19} can easily be handled in seconds to minutes. These figures give encouraging evidence that one can process moderate-sized datasets using the noisy-max mechanism. In our evaluation we consider only the simple two-party semi-honest model, for which we can achieve reasonable concrete efficiency. But since our main contribution is more efficient circuits, our improvements apply equally to multi-party and malicious models.

Discussion of Prior Work. The closest prior work is the celebrated result of Dwork *et al.* [14], which presents the idea of using secure computation protocols to implement differentially private processing of datasets *by the data owners themselves.* Indeed, our results in §3.2 make use of their observation that sampling Poisson and related distributions can be reduced to sampling several fair coins with different biases. Their paper also makes note of the inefficiency of standard sampling, however the approaches that they suggest to overcome the λ coin bottleneck have very large gate overheads.

Anandan and Clifton [1] present a two-party protocol based on homomorphic encryption to generate a single sample from a Laplace distribution in the presence of a malicious adversary. Their first protocol takes the approach of inverting the CDF and therefore is computationally expensive and was not implemented. They propose a second cut-and-choose style protocol that offers only polynomial security and report that 500 samples can be generated in 9 seconds.

Several prior works present tailored MPC protocols for specific differentially private algorithms. The problem of computing a differentially private sum was first considered by Dwork *et al.* [14] and has many follow-up works [2, 3, 11, 17, 34]. Shi *et al.* [34] also present a DP mechanism for computing sums that uses a single round, allows users to drop out, but does not match the accuracy achievable in the central model, and require a trusted setup phase. Pettai and Laud [32] use the sharemind MPC system to report on another implementation of the sum-and-aggregate mechanism for differentially private processing of counts, averages, medians, etc. These mechanisms are much simpler than report-noisy-max.

Eigner *et al.* [17] present PrivaDA as an architecture for distributed differential privacy that uses secure computation on floating point arithmetic to compute the distributed Laplace, the distributed discrete Laplace, and the distributed exponential mechanism. Their main technical contribution is to explain how to handle floating-point arithmetic, exponentiation and logarithm functions in secure computation, as well as converting between integer and floating representations. These operations are extremely complicated as secure computations; their experimental results for computing a single logarithm take 10s of seconds. In comparison, we are able to sample roughly 8000 geometric samples in the same time. As a result of these costs, they were unable to implement any full DP mechanisms. More concerning, Mironov [29] shows the hazards of using floating point approximations in differential privacy applications.

Several works have shown the *necessity* of secure computation (i.e. oblivious transfer) to achieve optimal accuracy without a trusted aggregator [20, 21, 27, 30]. Other work has considered securely implementing differentially private algorithms for gradient descent [7], continually monitoring sums [17, 33, 34], the private record-linkage problem [24], and heavy-hitters [9].

Algorithm	AND Gates	Random Bits
ODO-1 [14]	$O((\lambda + \log d)^2 \log d)$	2
ODO-2 [14]	$O(\lambda + \log d)$	$O(\lambda + \log d)$
ODO-3 [14]	$O(d(\lambda + \log d))$	2
ODO-4 [14]	$O((\lambda + \log d) \log(\lambda + d))$	2
MNM-1	$O(\log(\lambda + \log d))$	2
MNM-2	$O((\lambda + \log d) \log(\lambda + \log d))$	2

Table 1: Amortized $O(\cdot)$ cost per biased coin. The amortization is over d coins in total. We denote λ as the total statistical error for d coins. ODO-1, ODO-2, ODO-3, and ODO-4 are from [14] in the order they appear in that work starting at section 4.3. ODO-2 is the algorithm we implement due to its simplicity and low gate count. MNM refers to our coin flipping algorithm, and our numbering is 1 for the algorithm with asymptotic improvements and 2 for the algorithm used in practice.

2 SECURELY FLIPPING MANY COINS OF THE SAME BIAS

The fundamental problem we solve in this paper is to design a *boolean circuit* $C(d, p; \lambda)$ that can sample d coins of a bias p efficiently in both gates, communication, and number of random input bits required to perform the sampling with overall statistical difference $2^{-\lambda}$. The naïve circuit described in the introduction $C_0(d, p; \lambda)$ has amortized gate count $|C_0(d, p; \lambda)| = O(\log d + \lambda)$. Our circuit $C_{\text{mnm-1}}(d, p; \lambda)$ reduces this complexity to $O(\log(\log d + \lambda))$ by taking advantage of the expected two random bits needed per biased coin. Our algorithm does this by ending every comparison when the first difference in the p bias and stream of unbiased bits occurs. Informally, doing this privately requires the following functionalities:

(1) Sequential production of p's binary expansion
(2) A way to obliviously produce biased coins
(3) A method for reseting p's expansion obliviously

Note that all of these must be achieved within a secure computation, which does not provide many intuitive options for finishing random comparisons early while still being secure. We will describe how to acquire (1) and (3) in two ways later in the section. For (2) and one of those ways, we design oblivious data structures, which can store a number of coins at once, while providing operations to push and pop coins obliviously.

2.1 Oblivous data structures

The notion of an oblivious data structure was introduced by Goldreich and Ostrovsky [19] in the context of protecting the privacy of a CPU's memory access pattern against an adversary who can tap the memory channel bus. Subsequently several works have studied the overhead tradeoffs involved in implementing such data structures. The classical notion of security for oblivious data structures is stated in a RAM model and specified through the notion of a simulator and indistinguishably of the traces resulting from any two sequence of operations.

Instead of considering arbitrary RAM datastructures, we only consider a pair of very limited datastructures that support 1 and 2 operations. We only allow circuit-model implementations of these operations, and then evaluate the AND-gate complexity of these circuits as our measure of interest. This notion implies the standard simulator-based one for the limited scope and is consequently much simpler.

A *data structure* $D = (\mathcal{O}; B, C) = D(\mathcal{O})$ is a tuple consisting of a sequence of bits $B = b_1, \ldots, b_M$, a set of bookkeeping bits C,[3] and a fixed set of operators $\mathcal{O}$ which act on B and C. For the following let $C = \{c, r\}$, where c represents the current count of bits and r is a reset flag. We define three members (the ones relevant to our data structures) of the set of possible operators $\mathcal{O}^*$:

(1) $\mathsf{cpush}(f, D, b)$: returns $(B' = (b, b_1, \ldots, b_{M-1}), C' = \{c + 1, r\})$ if $f = 1$ and (B, C) otherwise. b represents the bit to be pushed.
(2) $\mathsf{creset}(f, D)$: returns $(B, C' = \{c, 1\})$ if $f = 1$ and (B, C) otherwise. In our construction, $r = 1$ denotes the need for a reset.
(3) $\mathsf{rpop}_{(\hat{B}, \hat{c})}(f, D)$: returns the bit $\hat{b}_1$ and $(\hat{B}' = (\hat{b}_2, \ldots, \hat{b}_M, 0)$, $C' = \{\hat{c} - 1, 0\})$ if $r = 1$, b_1 and $(B' = (b_2, \ldots, b_M, 0), C' = \{c - 1, 0\})$ otherwise. Here the values $\hat{B} = (\hat{b}_1, \ldots, \hat{b}_M)$, $\hat{c}$ are hard-coded values of the datastructure (typically, the initialized values before any operations).

[3]Conceptually, the bits in C can be included in the sequence of bits, but we separate them for convenience.

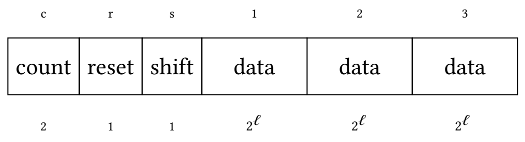


Figure 1: Depiction of the recursive data structure at level ℓ. The top row indicates how we name each field in the subsequent discussion. The bottom row indicates the field size in bits. Each level includes 4 bits of bookkeeping and 3 "buckets" that hold 2^ℓ bits each. Our implementation also includes a pointer to the next level for convenience, but this can be omitted if successive levels are arranged in memory as an array.

These operations take a conditional flag f as an input that determine whether the operation is performed or not. In the case of rpop, f is ignored in favor of an internal bit in C.

We consider boolean circuits that *implement* these operations on D. However, instead of requiring *uniform* circuits, we allow the circuit that implements the i^{th} operation on D *to depend* on i, i.e., the number of previous operations that have been applied to the data structure. The circuit that implements an operation cannot, however, depend on *the specific operations* that have been applied to D—only on the count. This extra ability allows scheduling "clean-up tasks" that simplify the datastructure at periodic intervals that are independent of the data being stored. We use the natural notion of *correctness* in which the circuit for each operation implements the semantics defined above.

Each of these circuits consist of boolean gates (AND and XOR), simple wires, and desigation of each wire as an input wire, an ouptut wire, or an internal wire. We measure the complexity of a circuit by counting the number of its AND gates.

We now proceed to describe our implementations of this data structure.

Construction. We use two data structures, both of which are essentially constructed as in Figure 1, and are hierarchical; level i of the structures contain a single bit to represent whether a level needs to be *reset*, two bits which store a count of the number of elements at this level, 3 data slots each of size 2^i bits, and finally a pointer to the next level of the data structure. The pointer is for convenience of notation and can be omitted in implementation by arranging the levels adjacent to one another in an array. The total capacity of the data structure is the sum of the sizes of the data slots at all of the levels. This design is inspired by the stack construction from [44]. Our first data structure, $D_{pop}(\mathcal{O}_{pop})$, is a data structure with $\mathcal{O}_{pop} = \{\mathsf{rpop}, \mathsf{creset}\}$, and follows Figure 1 precisely. Our second data structure is $D_{push}(\mathcal{O}_{push})$, with $\mathcal{O}_{push} = \{\mathsf{cpush}\}$, and it omits the reset bit from Figure 1.

All pushes and pops initially take place in level 0, but level 0 will become empty or full at different points during a sequence of stack operations. To address this, when level i is full it shifts some of its contents to level $i+1$ below, and when level i is empty, level $i+1$ shifts its contents to level i. To keep this operation oblivious, these shifts occur on a regular schedule: level i checks if it needs to make a shift every 2^{i+1} operations of a given type (push or pop). Notice this oblivious schedule ensures that overflows (or underflows) never occur except at possibly the last level (where they are ignored in our case). Thus, we use a shift circuit every second time a level is accessed, meaning shifts must be made not only when a level is empty/full but also when it could be empty/full after the next operation of a given type. The advantage of this shifting scheme is that, even though moving data twice as large is twice as expensive, level $i+1$ is accessed half as often as level i, so all levels have the same amortized cost. This makes the complexity per operation a favorable $O(\log n)$ for n element capacity, since a level 0 access has constant gate count and there are $O(\log n)$ levels total.

In contrast to the implementation in [44], our structures only support either push or pop operations, but not both. As a result, it suffices to have only 3 buckets per level (instead of 5), cutting down our gate count by a constant factor.

Oblivious Reset. Intuitively, we will be using D_{pop} to store bits of the binary expansion of some bias p and pop them off sequentially to give the functionality of (1). To achieve (3), we add a secret reset bit to each level of the stack which determines whether the level will set its slots to their original values of the datastructure before popping normally. After every oblivious reset, we set the reset bit to 0. When we pop the next bit of p's binary expansion and it is not equal to the next random bit, we set the reset bit of each level to 1 so that the next pop will start from the first bit of p's bias again.

Below we provide pseudo-code to more formally express the intuition above. We use the notation $\mathsf{mux}(f, a_0, a_1)$ to represent $a_0 + f \cdot (a_0 + a_1)$ where the operations are performed over F_2; in other words, this step returns a_f using $|a_0|$ AND gates. We show the pseudo-code for the pop operation first; the cpush operation is similar. The creset operation recursively sets the reset flag at each level of the hierarchy.

```
1:  procedure RPOP($f$, stk)            ▹ ret success bit $s$, data $d$
2:      $\mathsf{stk}.\{1,2,3,c\} \leftarrow \mathsf{mux}(\mathsf{stk}.r, \mathsf{stk}.\{1,2,3,c\}, \{\hat{1},\hat{2},\hat{3},\hat{c}\})$
        ▹ $\hat{x}$ is reset value of $x$
3:      $\mathsf{stk}.r \leftarrow 0$            ▹ always set reset bit to 0
4:      if $\mathsf{stk.next} \neq \perp$ then
5:          if $\mathsf{stk}.s = 1$ then
6:              $c_1 \leftarrow (\mathsf{stk}.c \overset{?}{\leq} 1)$
7:              $\mathsf{stk}.1 \leftarrow \mathsf{mux}(c_1, \mathsf{stk}.1, \mathsf{stk}.3)$
8:              $s, d \leftarrow \textsc{rpop}(c_1, \mathsf{stk.next})$
9:              $\mathsf{stk}.1, \mathsf{stk}.2 \leftarrow d$
10:             $\mathsf{stk}.c \leftarrow \mathsf{mux}(s \land c_1, \mathsf{stk}.c, \mathsf{stk}.c + 2)$
11:             $c_2 \leftarrow (\mathsf{stk}.c \overset{?}{\equiv} 0 \pmod 2)$
12:             $\mathsf{stk}.1 \leftarrow \mathsf{mux}(c_1 \land c_2, \mathsf{stk}.1, \mathsf{stk}.2)$
13:             $\mathsf{stk}.2 \leftarrow \mathsf{mux}(c_1 \land c_2, \mathsf{stk}.2, \mathsf{stk}.3)$
14:             $\mathsf{stk}.s \leftarrow 0$
15:         else
16:             $\mathsf{stk}.s \leftarrow 1$
17:         end if
18:     end if
19:     $s \leftarrow 1$     ▹ always pop (compare with rand bit) in our
    case
20:     $(d, \mathsf{stk}.c) \leftarrow (\mathsf{stk}.1, c - 1)$
21:     $\mathsf{stk}.1 \leftarrow \mathsf{stk}.2, \mathsf{stk}.2 \leftarrow \mathsf{stk}.3$
22:     return $(s, d)$
23: end procedure
```

```
1:  procedure CPUSH($f$, input, stk)          ▹ ret success bit $s$
2:      if $\mathsf{stk.next} \neq \perp$ then
3:          if $\mathsf{stk}.s = 1$ then
4:              $c_1 \leftarrow (\mathsf{stk}.c \overset{?}{\geq} 2)$
5:              $s' \leftarrow \mathsf{mux}(c_1, 0, \textsc{cpush}(c_1, \mathsf{stk.next}))$
6:              $\mathsf{stk}.3 \leftarrow \mathsf{mux}(s', \mathsf{stk}.3, \mathsf{stk}.1)$
7:              $\mathsf{stk}.c \leftarrow \mathsf{mux}(s', \mathsf{stk}.c, \mathsf{stk}.c - 2)$
8:              $\mathsf{stk}.s \leftarrow 0$
9:          else
10:             $\mathsf{stk}.s \leftarrow 1$
11:         end if
12:     end if
13:     $s \leftarrow \neg(\mathsf{stk}.c \overset{?}{=} 3)$                ▹ check fullness
14:     for $i = 1$ to $3$ do
15:         $\mathsf{stk}.i \leftarrow \mathsf{mux}(f \land (\mathsf{stk}.c \overset{?}{=} 3 - i), \mathsf{stk}.i, \mathsf{input})$
16:     end for
17:     $\mathsf{stk}.c \leftarrow \mathsf{mux}(f \land s, \mathsf{stk}.c, \mathsf{stk}.c + 1)$
18:     return $s$
19: end procedure
```

```
1:  procedure CRESET($f$, stk)                ▹ return nothing
2:      $\mathsf{stk}.r \leftarrow \mathsf{mux}(f, \mathsf{stk}.r, 1)$
3:      if $\mathsf{stk.next} \neq \perp$ then
4:          CRESET($f$, stk.next)
5:      end if
6:  end procedure
```

Analysis. We now state and prove the following theorem.

THEOREM 2.1. *Let data structure D have capacity n bits. The total number of AND gates required to implement n calls to* pop, creset *(respectively* cpush*) on D is $\Theta(n \log n)$.*

Consider a data structure that is designed to hold n bits. The i^{th} level of the data structure holds $3 \cdot 2^i$ bits, and therefore $k = O(\log n)$ levels are needed. Thus, it is easy to see that the creset operation on such a data structure requires $O(\log n)$ AND gates to implement since it performs one mux operation on a single bit per level. Each $\mathsf{mux}(\cdot, a_0, a_1)$ operation can be implemented using $|a_0|$ AND gates.

The analysis of pop is slightly more complicated but also require $O(n \log n)$ AND gates across n operations. Let $T(i)$ represent the number of AND gates required to implement a call of pop on level i of the hierarchy. When the shift bit at this level is 0, then only the AND gates from the mux operation in line 2 are required, and so $T(i) = 3 \cdot 2^i$. When shift is odd, then lines 7,10,12,13 contribue another $3 \cdot 2^i + 2$ gates, and the recursive call in line 8 contributes $T(i+1)$ gates. Over a sequence of n operations, the n calls to hierarchy level 0 contribute $n \cdot T(0)$ gates. Half of these calls require $3 \cdot 2^0$ gates, while the other $n/2$ calls require $3 \cdot 2^0 + (3 \cdot 2^0 + 3 + T(2))$ gates. Of these, $n/4$ terms of $T(2)$ add $3 \cdot 2^1$ gates, while the other $n/4$ contribute $3 \cdot 2^1 + (3 \cdot 2^1 + 3 + T(3))$. Expanding all such $T()$ terms and collecting, the total number of AND gates is

$$\sum_{i=1}^{k} \lceil n/2^i \rceil \cdot 3 \cdot 2^{i-1} + \lceil n/2^i \rceil (3 \cdot 2^i + 3)$$
$$\leq \sum_{i=1}^{k} \lceil n/2^i \rceil \left[3 \cdot 2^i + 3 \cdot 2^{i-1} + 3 \right]$$
$$\leq \sum_{i=1}^{k} 5n = O(n \log n)$$

An analysis of cpush is similar.

Discussion of Batching Parameters. Returning to the task of producing d biased coins of the same bias, we arrive at the issue of when to stop pushing coins onto the push-only stack. To use the least number of pushes, we could check if the stack is full before every push and stop when it is. However, this would potentially lose some privacy since we are revealing the total number of unbiased coined needed to make d coins. Additionally, stopping early does not mesh with the constraint of a static circuit. Given that it is unknown how many pushes will be needed to generate a group of d coins, we choose a small constant c such that the chance of needing more than cd pushes to make d biased coins is less than $2^{-\lambda}$. If we assume the stack is full after cd pushes, we can also empty out the stack for free by simply wiring the slots to d coins. The final question is to choose what size stack will be used to make a total of d coins. The obvious choice is to use a stack large enough to store all d coins at once, which would also minimize c. However, by making coins in batches of some size $g < d$ we can reduce the number of levels and thus the amortized cost per push of the stack, while increasing c very marginally, reducing our complexity. The process for choosing c and g is described thoroughly in §5.

2.2 Make-Batch 1

In this section, we describe our first method for producing a batch of g biased coins via $c \cdot g$ cpush operations on a push-only stack. This method for MAKE-BATCH yields an asymptotic complexity which clearly dominates that of ODO-2 for making d coins, but does not win in practice until $\lambda + \log d$ (union for overall statistical difference) is large for practical standards. This MAKE-BATCH uses a pop-only resettable stack described above to achieve properties (1) and (3). Informally, the loop mimics the "lazy sampling" method in which the binary expansion of the bias p is compared bit-by-bit with fair random coins. As soon as the random coin differs from a bit of the expansion, the loop pushes a new sample onto a stack that collects samples. Each iteration of the loop thus consists of one pop operation, one cpush operation, and one creset operation in case of success. The pseudocode is as follows:

```
1:  Let rstack_p be the resettable, pop-only stack that contains
    a given p
2:  Let RPOP(rstack_p) be a pop that is preceded by a reset if
    the reset bit is 1
3:  Let cstack be the push-only stack of size g
4:  procedure MAKE-BATCH(c, g, p)
5:      for w = 1 to cg do
6:          b ←NEXT(1, coins)                   ▹ next fair bit
7:          t ←RPOP(rstack_p)                   ▹ next bias bit
8:          f ← b ⊕ t                 ▹ f = 1 if difference found
9:          CPUSH(f, ¬b, cstack)        ▹ push ¬b if cstack has
    room and f = 1
10:         CRESET(f, rstack_p)                 ▹ oblivious reset
11:     end for
12: end procedure
```

Theorem 2.2. *Let λ be the security parameter and d be the total number of coins. Then the amortized circuit complexity of MAKE-BATCH for making d coins is $O(\log(\lambda + \log d))$.*

2.3 Make-Batch 2

Our second algorithm for MAKE-BATCH does better than the first when $\lambda + \log d$ is less extreme, which is generally the case in practice. Instead of using a stack for satisfying property 1, a predicate function is constructed to take an integer j as input and return the jth bit of some probability p. This predicate function is used in conjunction with a counter that tracks what j should be at a given step. To satisfy 3, we reset this counter depending on the XOR of the next unbiased bit and the next bit of p. We present the pseudocode below:

```
1:  Let GET(p, j) be the predicate function that gets the jth bit
    of a binary expansion p
2:  procedure MAKE-BATCH(c, g, p)
3:      count ← 0
4:      for w = 1 to cg do
5:          b ←NEXT(1, coins)                   ▹ next fair bit
6:          t ←GET(p, count)                    ▹ next bias bit
7:          f ← b ⊕ t                 ▹ f = 1 if difference found
8:          CPUSH(f, ¬b, cstack)
9:          count ← count + 1
10:         if f = 1 then
11:             count ← 0
12:         end if
13:     end for
14: end procedure
```

Theorem 2.3. *Let λ be the security parameter and d be the total number of coins. Then the amortized circuit complexity of MAKE-BATCH for making d coins is $O((\lambda + \log d) \log(\lambda + \log d))$.*

3 REPORT-NOISY-MAX APPLICATION

In this section we demonstrate how we can use our new methods for batch-sampling biased coins to securely implement one of the foundational algorithms in differential privacy (DP), the *report-noisy-max mechanism* [4] (a variant of the widely known *exponential mechanism*). We begin by recalling the definition of DP.

Definition 3.1 ([15]). *Let $\mathcal{X}$ be the universe of possible dataset entries, $\mathcal{R}$ be a range of outputs, and $\varepsilon, \delta \geq 0$ be parameters. We say a randomized algorithm $\mathcal{A} : \mathcal{X}^n \rightarrow \mathcal{R}$ is (ε, δ)-differentially private if for every two datasets $x = (x_1, \dots, x_i, \dots, x_n) \in \mathcal{X}^n$ and $x' = (x_1, \dots, x_i', \dots, x_n) \in \mathcal{X}^n$ that are the same except for one individual's data, and for every set of outcomes $S \subseteq \mathcal{R}$, we have $\mathbb{P}[\mathcal{A}(x) \in S] \leq e^{\varepsilon}\mathbb{P}[\mathcal{A}(x') \in S] + \delta$.*

We typically view ε as the "privacy level" and require it to be a small constant, as having ε too small leads to poor utility, and leaving it too large provides meaningless privacy. For example, Google's RAPPOR [18] and PROCHLO [5] use $\varepsilon = \ln 3$ and $\varepsilon = 2.25$, respectively. We think of δ as a "failure probability" for the algorithm, and require that it be "cryptographically small", e.g. 2^{-80}.

3.1 Report-Noisy-Max

One of the most useful differentially private algorithms is the *report-noisy-max* mechanism [4] (see [16] for a textbook treatment). This mechanism is a more practical implementation of the widely known *exponential mechanism* [28], and the two mechanisms solve the same problem with identical privacy and utility guarantees.

Given a dataset $\mathcal{X}^n$ and discrete set of choices $\mathcal{Y}$ (denote $|\mathcal{Y}| = d$), as well as a utility function $u \colon \mathcal{X}^n \times \mathcal{Y} \rightarrow \mathbb{R}$ such that $u(x, y)$ is the utility of choice $y \in \mathcal{Y}$ on dataset $x \in \mathcal{X}^n$, a user would naturally want to select a $y \in \mathcal{Y}$ that has high utility on the given dataset. For example, $\mathcal{Y}$ might be a set of classifiers for a machine learning model, and $u(x, y)$ might be the number of examples in the dataset that y classifies correctly. The report-noisy-max mechanism is a way to privately select an element $\hat{y}$ such that $u(x, \tilde{y}) \geq \max_{y \in \mathcal{Y}} u(x, y) - O(\log d)$. The importance of this mechanism comes from the fact that the error grows only logarithmically in the number of choices.

The report-noisy-max algorithm works in two steps: First, *securely* compute noisy scores $\hat{u}_y = s(x, y) + z_y$ for each $y \in \mathcal{Y}$, where z_y is a suitably chosen random variable typically Laplace or geometric. Second, return $\tilde{y}$ that maximizes the noisy score $\hat{u}_{\tilde{y}}$. It is crucial for privacy that the intermediate noisy scores are not revealed, only the final choice $\tilde{y}$. For our purposes, we draw from geometric noise in the first step. We label the noisy max mechanism that adds $\text{Geo}(2/\varepsilon)$ (the discrete version of $\text{Lap}(2/\varepsilon)$) to each score as $\text{NM-Geo}(2/\varepsilon)$. Since we are drawing samples from the geometric distribution, we restrict the output space of our utility function to $u \colon \mathcal{X}^n \times \mathcal{Y} \rightarrow \mathbb{Z}$. We note that rounding down scores to integers for utility functions that have output space $\mathbb{R}$ increases the error by at most 1, which is small in comparison to $\log d$. Since this mechanism can be implemented by sampling many independent noise variables, each of which require sampling many biased coins, it is ideally suited to our methods.

Theorem 3.2 ([16, 40]). *$\text{NM-Geo}(2/\varepsilon)$ is $(\varepsilon, 0)$-differentially private.*

3.2 Review: Sampling Exponential Noise via Poisson

In this section, we review the techniques from [14] showing how to sample the Poisson distribution in order to approximate the exponential distribution. Recall that the celebrated Poisson distribution is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval if these events occur with a known constant rate λ. For example, such a distribution can model the number of soldiers in the Prussian army killed accidentally by horse kicks [41]. Specifically, the support of the Poisson distribution are the non-negative integers $0, 1, 2, \dots$, and the probability mass function is defined as $f(k; \lambda) = Pr[X = k] = \frac{\lambda^k e^{-\lambda}}{k!}$. As in [14], we sample from this distribution in order to approximate the exponential distribution.

Naive methods. Generically, one can sample any function with cumulative distribution function ρ by first sampling $r \in [0, 1]$ and then finding the maximum x such that $r < cdf(x)$. The latter maximization problem can be solved by inverting the CDF. Thus, in the case of drawing Poisson or exponential noise, the complexity of this naive sampling approach will be dominated by the complexity of computing $\ln x$ (which appears in the inverse CDF).

Bitwise sampling. The main observation in [14] is that the special structure of an exponential distribution enables the generation of the binary representation of an exponential variable using a number of coins that is independent of the bias. Thus, by calling the noise sample some κ bit number, one can compute the probability that bit i of a sample is 0 or 1 as seen in [14]. This bounds the distribution to the interval $(-2^{\kappa}, 2^{\kappa})$, since after generating a κ bit noise sample we flip a fair coin to choose whether the noise is negative or positive, as we desire two-sided exponential noise. Since $Pr[X = x] \propto \exp(-|x|/R)$ in the exponential distribution (with scaling constant R), the probability that bit κ is 1 diminishes at a doubly exponential rate, meaning κ will stay nearly constant as the number of samples and privacy requirements grow. We note that sampling in this way implicitly makes the noise an integer, meaning we are actually sampling from the geometric distribution. This is ideal since geometric noise satisfies our differential privacy needs while avoiding the expensive computation of the natural logarithm.

3.3 Implementing Report-Noisy-Max

Using bitwise sampling from §3.2 and make-batch from §2, we can construct a secure implementation of noisy max. We present the pseudocode below:

```
1: Let $\lambda$ be the security parameter, $\varepsilon$ be the DP parameter,
   and $\kappa$ restrict our noise domain to $(-2^{\kappa}, 2^{\kappa})$
2: Let $p_0, p_1, \dots$ be the binary expansions (out to $f$ bits, de-
   rived from $\varepsilon$) for the biased coins needed to compute $k$ bit
   noise, where $k \leq \kappa$ is derived from $\kappa$ to optimize run time
3: Let cstack be the push-only stack of size $g$ used for making
   batches of $g$ coins, where $g$ is picked along with a small
   constant $c$ to optimize run time
4: procedure $\mathsf{MNM}_{\lambda,\varepsilon,\kappa}(u_1, \dots, u_d)$        ▷ $u_i = u(x, y_i)$
```

```
5:      for $i = k$ to 0 do
6:          for $j = 1$ to $d/g$ do
7:              MAKE-BATCH$(c, g, p_i)$
8:              $s_1, \ldots, s_g \leftarrow$PURGE(cstack)      ▷ output coins
9:              $n_{g(j-1)+1}, \ldots, n_{gj} \leftarrow n_{g(j-1)+1}|s_1, \ldots, n_{gj}|s_g$
    ▷ concat noise, sample $n_i$ corresponds to $u_i$
10:         end for
11:     end for
12:     return MAX-IDX$(u_1 \pm n_1, \ldots, u_d \pm n_d)$
13: end procedure
```

We note that the pseudocode for $\mathsf{ODO}_{\lambda,\varepsilon,\kappa}$, the algorithm that uses comparator circuits to flip all biased coins, follows directly from the bitwise sampling in §3.2 and the definition of noisy max, so we do not provide it.

3.4 Complexity Theorems

We now present the following theorems which follow very simply from the MAKE-BATCH theorems:

THEOREM 3.3. *Let $\varepsilon \in [0.001, 10]$ and the number of bits for all u_i (potentially padded) be constants, λ be as above, d be the number of choices, and $\kappa = O(\log(\lambda + \log d))$. Then the circuit complexity of $\mathsf{MNM}_{\lambda,\varepsilon,\kappa}$ with MAKE-BATCH from §2.2 is $O(d \log^2(\lambda + \log d))$.*

THEOREM 3.4. *Let $\varepsilon \in [0.001, 10]$ and the number of bits for all u_i (potentially padded) be constants, λ be as above, d be the number of choices, and $\kappa = O(\log(\lambda + \log d))$. Then the circuit complexity of $\mathsf{MNM}_{\lambda,\varepsilon,\kappa}$ with MAKE-BATCH from §2.3 is $O(d(\lambda + \log d) \log^2(\lambda + \log d))$.*

THEOREM 3.5. *Let $\varepsilon \in [0.001, 10]$ and the number of bits for all u_i (potentially padded) be constants, λ be as above, d be the number of choices, and $\kappa = O(\log(\lambda + \log d))$. Then the circuit complexity of $\mathsf{ODO}_{\lambda,\varepsilon,\kappa}$ is $O(d \log(\lambda + \log d)(\lambda + \log d))$.*

3.5 Proof of Differential Privacy

Let $\mathcal{M} : \mathcal{X}^n \to \mathcal{Y}$ be the noisy max algorithm using geometric noise with finite domain $(-2^\kappa, 2^\kappa)$ such that:

$$\mathcal{M}(x) = \begin{cases} \text{NM-Geo}(2/\varepsilon) & \text{w.p. } 1-\delta \\ \mathcal{F}(x) & \text{w.p. } \delta \end{cases},$$

where $\mathcal{F}$ is the function executed when a sample is out of the range $(-2^\kappa, 2^\kappa)$ and NM-Geo$(2/\varepsilon)$ is the "perfect" noisy max algorithm defined as above. By the privacy of NM-Geo$(2/\varepsilon)$, $\mathcal{M}$ is (ε, δ)-differentially private. Let $\widetilde{\mathcal{M}} : \mathcal{X}^n \to \mathbb{N}$ be MNM, which is the same as $\mathcal{M}$ except for the possibilities of a biased coin failing and/or that cg pushes (for c and g as in the pseudocode) create less than g coins. We define $\widetilde{\mathcal{M}}$ like so:

$$\widetilde{\mathcal{M}}(x) = \begin{cases} \mathcal{M}(x) & \text{w.p. } 1-(\rho+\nu) \quad (\neg E) \\ \mathcal{G}(x) & \text{w.p. } \rho+\nu \quad (E) \end{cases},$$

where E is the event in which any biased comparator fails (represented by ρ) and/or any set of cg pushes fails to produce g coins (represented by ν). We let $\mathcal{G}$ be the function executed when E happens.

THEOREM 3.6. *$\widetilde{\mathcal{M}}$ is $(\varepsilon, \delta + \rho + \nu)$-differentially private.*

Here we have

$$\delta = 2e^{-(2^\kappa \varepsilon - \ln d)}$$

$$\rho = 2^{-\lambda}, \text{ and}$$

$$\nu \le \frac{d}{g} \exp\left(-2\frac{\left(\frac{cg}{2} - (g-1)\right)^2}{cg}\right) \le 2^{-\lambda}$$

for some choice of κ, ε, λ, c, and g, which are defined the same way as in the pseudocode. We choose c and g given λ such that the last inequality holds. Note that ODO can be proven $(\varepsilon, \delta + \rho)$-differentially private in the same way.

Discussion. Since our algorithm introduces three addends that sum to δ, its a good idea to set $\lambda = \log(4/\delta)$ for the goal of (ε, δ)-DP. This will make ν and ρ both less than or equal to $\delta/4$, making their sum less than or equal to $\delta/2$. If κ is set such that $\delta' = 2e^{-(2^\kappa \varepsilon - \ln d)} \le \delta/2$ as well (which happens when $\kappa = O(\lambda + \log d)$), we have $\delta' + \rho + \nu \le \delta$. Choosing κ and λ this way yields a complexity of $O(d \log^2 \log(d/\delta))$ for the first MAKE-BATCH, and $O(d \log(d/\delta) \log^2 \log(d/\delta))$ for the second. Since ε is a parameter to the algorithm that determines the exact biases for computing the bits of noise, it is already attained. It is worth noting however, that smaller ε for the same desired δ may increase the algorithm's complexity slightly.

4 OPTIMIZATIONS

In this section we will describe some of the additional methods used to further cut down gates when implementing MNM.

Special Values of ε. Our primary reduction comes from choices of ε that have especially easy biases to produce, namely ε of the form $\varepsilon = 2^{-i} \ln 2$ for $i \in \mathbb{Z}$. When ε takes this form, the expression for the probability of bit j being 1 is reduced from

$$1/(1 + \exp(2^{j-1}\varepsilon)) \text{ to } 1/(1 + 2^{2^{j-i-1}}).$$

When the expression $j - i - 1 \ge 0$, we have a fully periodic binary expansion for the probability that bit j is 1 (e.g. 01010101... for $j - i - 1 = 0$). This allows us to produce the binary expansion for $\mathbb{P}[\text{bit } j = 1]$ by simply taking bit $j - i - 1$ of count (see §2.3), making GET a 0 gate function! Thus, if one does not have precise needs for ε, one can find the first expression of the form $2^{-i} \ln 2$ less than their approximate ε threshold and have a number of periodic expansions among the k biases flipped in the MNM protocol.

Remainder Batch. A lesser reduction we use is when finding the optimal c and g for batches, having the last group be a potentially smaller size, in order to make the least amount of total coins possible. With a total of d coins to make, this is done by simply taking $w = d \pmod g$ and finding the least expression of the form $3(2^i) > w$, which is then used as the size of the final group.

5 EVALUATION

The main contribution of this paper is the design of a new circuit family for sampling biased coins that is suitable for use in secure computation protocols. To illustrate the benefits of this new design, we have implemented our new sampling schemes, the ODO sampling scheme, and the report-noisy-max mechanism. The focus of the paper is not on secure computation, and therefore we consider the simpler two-party honest-but curious model; our techniques, however, apply equally to multi-party computation protocols that handle a variety of adversarial models.

Implementation Details. Our code can be found at https://www.gitlab.com/neucrypt/securely_sampling. We implemented and benchmarked both ODO and MNM, using Obliv-C [45], an extension of C that compiles and executes Yao's Garbled Circuits protocols with many protocol-level optimizations.

Benchmarks were performed using Ubuntu 18.04 with Linux kernel 4.18.0-1009-gcp 64-bit, running on pairs of identical Google Cloud Instance `n1-highcpu-4` instances. Code was compiled using `gcc version 8.2.0 (Ubuntu 8.2.0-7ubuntu1)`, with the `-O3 -march=native` flags.

We evaluated performance in two network settings. In the first network setting that mimics a LAN setup, all instances ran in the same `us-east1-b` datacenter. Using `iperf`, we measured the bandwidth between the pairs of instances to be 7.5 gigabits per second and the ping times to be 0.4ms. The second network setting reflects a typical WAN in which one machine was in the `us-east1` datacenter while the others were in the `us-west1` datacenter. Again using `iperf`, we measured the bandwidth between the two instances to be 330 megabits per second. These two network settings highlight the difference in network communication requirements between the various algorithms.

Selection of parameters. Using our MNM sampler requires choosing the following parameters:

(1) u: This parameter represents the number of pushes (cg as described above) needed to produce g coins with a desired chance of failure. In our experiments for $\varepsilon = 2^{-3}\ln 2, \delta = 2^{-60}$, this parameter ranged from 1941 to 6947.
(2) g: The primary batch size used to make all groups except for the remainder group (which in some cases is still size g). In our experiments for $\varepsilon = 2^{-3}\ln 2, \delta = 2^{-60}$, this parameter ranged from 765 to 3069.
(3) ℓ: The length of the bias for a desired $2^{-\lambda}$ statistical difference overall (f in the MNM pseudocode). In our experiments for $\varepsilon = 2^{-3}\ln 2, \delta = 2^{-60}$, this parameter ranged from 78 to 85.
(4) v: This represents the number of pushes needed to produce q coins with the same desired chance of failure as each of the batches of g. In our experiments for $\varepsilon = 2^{-3}\ln 2, \delta = 2^{-60}$, this parameter ranged from 1066 to 6947.
(5) q: The remainder batch size, used as an optimization (to make as few extra coins as possible). In our experiments for $\varepsilon = 2^{-3}\ln 2, \delta = 2^{-60}$, this parameter ranged from 381 to 3069.

In choosing these parameters, we picked κ as in our differential privacy discussion thus letting us solve for k as described in Theorem 3.3. Then we iterated over the choices for g, which are $3(2^i)$ for $i = 0, 1, 2, \ldots, 15$ (for $i > 15$, the cost per operation is too high compared to the minor reduction of c). For each g, we found the minimum number of pushes needed to make

$$\mathbb{P}[cg \text{ pushes yield } < g \text{ coins}] \leq 2^{-(f - \log g)},$$

with $f \geq \lambda + \log\kappa + \log d$. By taking $d \pmod g$ we could easily deduce what the remainder group would be, and the number of pushes needed for that to satisfy our desired overall chance of failure. With this done, we calculated what the total concrete gate count would be for noisy max based on our benchmarks of data structure operations and the cost of evaluating a $\log f$ bit predicate using multiple efficient 6-bit predicates. When doing this we first compared whether the pop-only stack or the predicate would be faster and chose the appropriate one. Finally, we took the parameters that yielded the lowest estimated concrete gate count for noisy max.

λ	pop Method	Predicate method
64	43.6	16
128	52.2	24
192	60.8	32
256	60.8	39
320	60.8	47
384	69.3	54
448	69.3	61
512	**69.3**	**69**
576	69.3	77
640	69.3	84

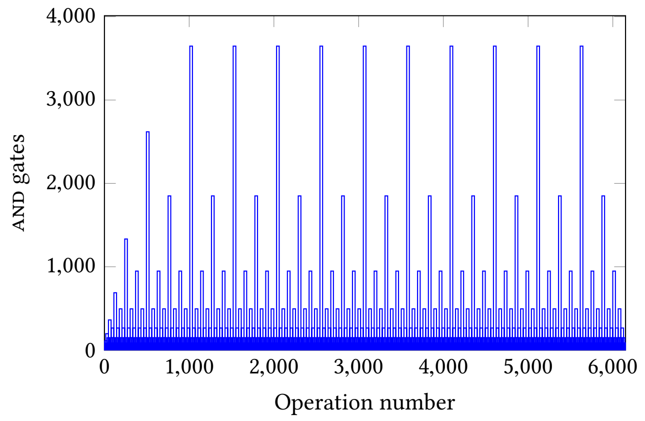
Table 2: Amortized number of AND gates for pop vs predicate as the length of the bias λ increases. For pop we took the average of 10000 iterations of calling pop and conditionally resetting on a random bit. The crossover point is roughly $\lambda > 512$ which is a highly secure setting, but certainly a reasonable parameter setting.

5.1 Microbenchmarks of datastructures

In this section we present the gate complexity of our cpush, pop, and creset operations, as well as the complexity of our predicate implementations for different biases. To compute these, we modified our Obliv-C implementation to report specific gate counts.

Complexity of push. Here we empirically measure the gate complexity of our cpush implementation. We consider stacks of size $n = 3 \cdot 2^t$ bits and then apply n conditional cpush operations while measuring the number of gates required for each operation. Figure 2 graphs the number of gates for the first 6141 operations as well the average number of gates for the first i operations.

Crossover for pop*.* In the section we compare the performance of our predicate versus the pop operation for producing the j^{th} bit of the binary expansion of a bias p. We compute the average number of gates required for pop operations, and the size of our predicate solution for increasingly long binary expansions of the bias. Our data is summarized in Table 2.



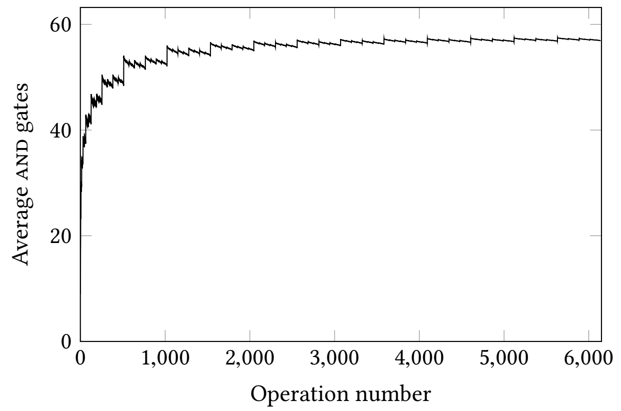


Figure 2: The top plot shows the exact number of AND gates in i^{th} cpush operation. The bottom plot show a running average number of gates for the first i operations, which fits closely the amortized $O(\log n)$ complexity we analyze.

5.2 Two Party d-Sample Benchmarks

We benchmarked the action of generating d samples from $\text{Geo}(2/\varepsilon)$ using the second version of MAKE-BATCH, which uses a predicate function to generate the bias. For comparison purposes, we also benchmarked the ODO implementation. For both implementations we varied the number of samples to make between 2^{12} and 2^{19}. We also sampled with two different ε: one of the form $\varepsilon = 2^{-i} \ln 2$ ($\varepsilon = 2^{-3} \ln 2$), and one not in that form ($\varepsilon = 0.1$). For each value of ε, we benchmarked for $\delta = 2^{-60}$ and $\delta = 2^{-80}$. We recorded the wall-clock time for the two aforementioned network settings and present our results for this in Figures 3a and 3c. The total number of bytes transmitted among both parties and the sum of the number of non-free Yao gates and the number of unbiased coins used are shown in Figures 3b and 3d, respectively. We note that cost and communication are static across different networks.

As we expected, our protocol scales very well with d in all categories. Despite the asymptotic behavior with the second MAKE-BATCH being sub-optimal, it is understandable that it scales well, as for lower values of $\lambda + \log d$ the cost of the predicate function is roughly constant, meaning the complexity is just as good as our protocol with the first MAKE-BATCH.

5.3 Two Party Noisy Max Benchmarks

Next we report on our full implementation of the noisy max algorithm using our improved biased coin sampling procedure. We expect the performance for noisy-max to be dominated by the cost of the sampling, and the data below supports this claim. In our two-party setup, we have each party contribute half of the dataset. We vary the size of the dataset from $2^{12} = 4096$ to 2^{19}, using 32-bit integer entries for the data. The benchmarks are run with 2 machines running in the same `us-east1` datacenter. The results are presented in Table 3.

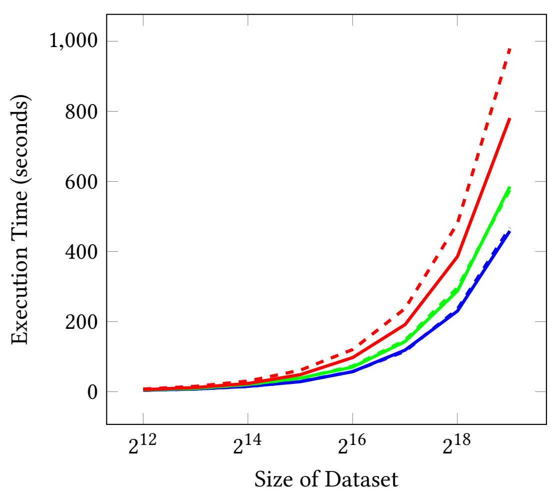
As predicted by our analysis, the cost grows slowly between $\delta = 2^{-60}$ and $\delta = 2^{-80}$; at $d = 2^{19}$, the difference is only 10s or 2%. We note that the communication overhead is quite high but feasible for moderate-sized domains.

δ	d	AND gates	Comm (10^6 b)	Time (s)
2^{-60}	4096	8,349,483	340.3	4.40
	8192	16,454,933	670.6	8.11
	16384	32,751,039	1335.3	17.87
	32768	64,584,144	2632.7	31.32
	65536	129,371,034	5271.8	63.53
	131072	259,005,597	10554.2	126.31
	262144	515,833,031	21020.8	242.69
	524288	1,033,115,150	42099.7	488.05
2^{-80}	4096	8,613,824	351.0	4.78
	8192	16,841,275	686.2	8.91
	16384	33,408,111	1360.9	16.07
	32768	66,031,953	2691.0	32.45
	65536	131,256,973	5347.8	62.46
	131072	262,730,472	10704.4	124.87
	262144	523,257,767	21320.0	254.11
	524288	1,047,606,374	42683.8	493.63

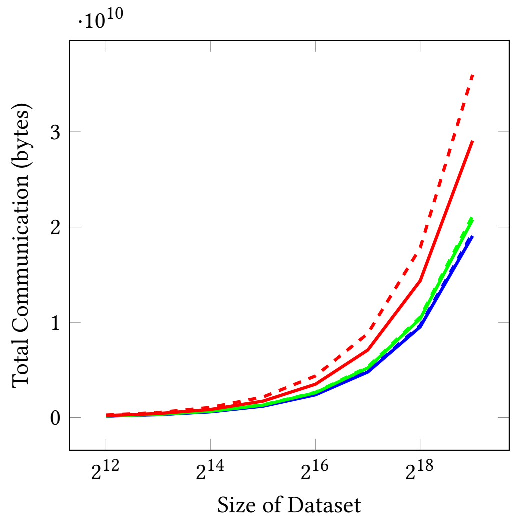
Table 3: Summary of costs for running report-noisy-max mechanism with $\varepsilon = 2^{-3} \ln 2 = 0.0866$ on datasets of increasing size and $\delta \in \{2^{-60}, 2^{-80}\}$.

ACKNOWLEDGMENTS

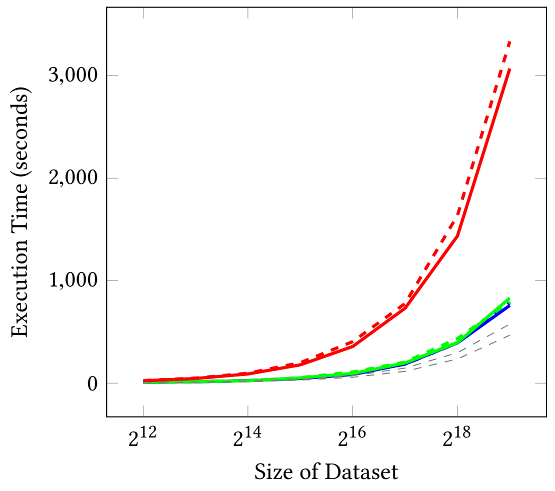
The authors were supported by NSF grant CNS-1816028. Part of this work was done while JU was visiting the Simons Institute for Theory of Computing. We thank Jack Doerner for his help with the implementation.



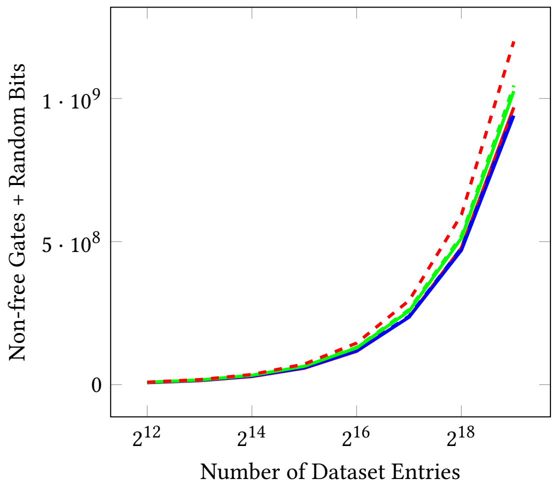
(a) d-Sample Wall-clock Time, east-east



(b) d-Sample Communication



(c) d-Sample Wall-clock, east-west



(d) d-Sample Yao Gates + Random Bits

🟦 MNM ($\varepsilon = 2^{-3} \ln 2$) 🟩 MNM ($\varepsilon = 0.1$) 🟥 ODO ($\varepsilon = 0.1, 2^{-3} \ln 2$)
—— $\delta = 2^{-60}$ - - - $\delta = 2^{-80}$

Figure 3: d-Sample Benchmark Results. We measure time and communication to produce d samples from Geo$(2/\varepsilon)$ in two network settings. In these graphs, d varies for two choices of ε and two δ for each ε. We note that ODO does not change based on the form of ε, so we use one plot for the two values of ε. Across all of these parameters, the MNM technique dominates the ODO. In graph (c), the gray lines represent the same MNM performance lines from graph (a) for comparison purposes.

REFERENCES

[1] Balamurugan Anandan and Chris Clifton. 2015. Laplace noise generation for two-party computational differential privacy. In *2015 13th Annual Conference on Privacy, Security and Trust (PST)*.

[2] Gilles Barthe, George Danezis, Benjamin Grégoire, César Kunz, and Santiago Zanella-Beguelin. 2013. Verified computational differential privacy with applications to smart metering. In *2013 IEEE 26th Computer Security Foundations Symposium*. IEEE, 287–301.

[3] Amos Beimel, Kobbi Nissim, and Eran Omri. 2008. Distributed private data analysis: Simultaneously solving how and what. In *Annual International Cryptology Conference*. Springer, 451–468.

[4] Raghav Bhaskar, Srivatsan Laxman, Adam Smith, and Abhradeep Thakurta. 2010. Discovering frequent patterns in sensitive data. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 503–512.

[5] Andrea Bittau, Ulfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Usharsee Kode, Julien Tinnes, and Bernhard Seefeld. 2017. PROCHLO: Strong Privacy for Analytics in the Crowd. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*.

[6] Avrim Blum, Katrina Ligett, and Aaron Roth. 2013. A learning theory approach to noninteractive database privacy. *J. ACM* 60, 2 (2013), 12.

[7] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical Secure Aggregation for Privacy Preserving Machine Learning. *IACR Cryptology ePrint Archive* (2017).

[8] Çagdas Çalik, Meltem Sönmez Turan, and René Peralta. 2018. The Multiplicative Complexity of 6-variable Boolean Functions. (2018).

[9] T.-H. Hubert Chan, Mingfei Li, Elaine Shi, and Wenchang Xu. 2012. Differentially Private Continual Monitoring of Heavy Hitters from Distributed Streams. In *Privacy Enhancing Technologies - 12th International Symposium, PETS 2012, Vigo, Spain, July 11-13, 2012. Proceedings*. 140–159.

[10] T-H Hubert Chan, Elaine Shi, and Dawn Song. 2011. Private and continual release of statistics. *ACM Transactions on Information and System Security (TISSEC)* 14, 3 (2011), 26.

[11] Ruichuan Chen, Alexey Reznichenko, Paul Francis, and Johanes Gehrke. 2012. Towards statistical queries over distributed private user data. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 169–182.

[12] Jack Doerner and abhi shelat. 2017. Scaling ORAM for Secure Computation. In *ACM CCS'17*.

[13] John C Duchi, Michael I Jordan, and Martin J Wainwright. 2013. Local privacy and statistical minimax rates. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*. IEEE, 429–438.

[14] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *EUROCRYPT*.

[15] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography Conference (TCC)*.

[16] Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science* 9, 3-4 (2014), 211–407.

[17] Fabienne Eigner, Aniket Kate, Matteo Maffei, Francesca Pampaloni, and Ivan Pryvalov. 2014. Differentially private data aggregation with optimal utility. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 316–325.

[18] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *ACM Conference on Computer and Communications Security (CCS)*.

[19] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM* 43, 3 (1996).

[20] Vipul Goyal, Dakshita Khurana, Ilya Mironov, Omkant Pandey, and Amit Sahai. 2016. Do Distributed Differentially-Private Protocols Require Oblivious Transfer?. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*.

[21] Vipul Goyal, Ilya Mironov, Omkant Pandey, and Amit Sahai. 2013. Accuracy-Privacy Tradeoffs for Two-Party Differentially Private Protocols. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*.

[22] Samuel Haney, Ashwin Machanavajjhala, John M Abowd, Matthew Graham, Mark Kutzbach, and Lars Vilhuber. 2017. Utility Cost of Formal Privacy for Releasing National Employer-Employee Statistics. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1339–1354.

[23] Moritz Hardt, Katrina Ligett, and Frank McSherry. 2012. A Simple and Practical Algorithm for Differentially Private Data Release. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. 2348–2356.

[24] Xi He, Ashwin Machanavajjhala, Cheryl Flynn, and Divesh Srivastava. 2017. Composing Differential Privacy and Secure Computation: A case study on scaling private record linkage. *arXiv preprint arXiv:1702.00535* (2017).

[25] Shiva Prasad Kasiviswanathan, Homin K. Lee, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. 2008. What Can We Learn Privately?. In *Foundations of Computer Science (FOCS)*. IEEE.

[26] Steve Lu and Rafail Ostrovsky. 2014. Garbled RAM Revisited, Part II. Cryptology ePrint Archive, Report 2014/083.

[27] Andrew McGregor, Ilya Mironov, Toniann Pitassi, Omer Reingold, Kunal Talwar, and Salil Vadhan. 2010. The limits of two-party differential privacy. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*. IEEE, 81–90.

[28] Frank McSherry and Kunal Talwar. 2007. Mechanism Design via Differential Privacy. In *IEEE Foundations of Computer Science (FOCS)*.

[29] Ilya Mironov. 2012. On significance of the least significant bits for differential privacy. In *Proceedings of the 2012 ACM Conference on Computer and cCommunications Security (CCS)*. ACM.

[30] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. 2009. Computational differential privacy. In *Advances in Cryptology-CRYPTO 2009*. Springer, 126–142.

[31] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. 2018. PanORAMa: Oblivious RAM with logarithmic overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 871–882.

[32] Martin Pettai and Peeter Laud. 2015. Combining Differential Privacy and Secure Multiparty Computation. In *ACSAC 2015*. ACM, New York, NY, USA, 421–430. https://doi.org/10.1145/2818000.2818027

[33] Vibhor Rastogi and Suman Nath. 2010. Differentially private aggregation of distributed time-series with transformation and encryption. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 735–746.

[34] Elaine Shi, T.-H. Hubert Chan, Eleanor G. Rieffel, Richard Chow, and Dawn Song. 2011. Privacy-Preserving Aggregation of Time-Series Data. In *Proceedings of the Network and Distributed System Security Symposium, (NDSS) 2011*.

[35] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O((logN)3) Worst-Case Cost. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*. 197–214.

[36] Kunal Talwar, Abhradeep Thakurta, and Li Zhang. 2015. Nearly optimal private LASSO. In *Advances in Neural Information Processing Systems, NIPS*. 3025–3033.

[37] Abhradeep Guha Thakurta, Andrew H Vyrros, Umesh S Vaishampayan, Gaurav Kapoor, Julien Freudiger, Vivek Rangarajan Sridhar, and Doug Davidson. 2017. Learning new words. US Patent 9,645,998.

[38] Abhradeep Guha Thakurta, Andrew H Vyrros, Umesh S Vaishampayan, Gaurav Kapoor, Julien Freudinger, Vipul Ved Prakash, Arnaud Legendre, and Steven Duplinsky. 2017. Emoji frequency detection and deep link frequency. US Patent 9,705,908.

[39] Jonathan Ullman. 2018. Tight lower bounds for locally differentially private selection. *arXiv preprint arXiv:1802.02638* (2018).

[40] Salil Vadhan. 2016. The complexity of differential privacy. *http://privacytools.seas.harvard.edu/publications/complexity-differential-privacy* (2016).

[41] Ladislaus von Bortkiewicz. 1898. Das Gesetz der kleinen Zahlen [The law of small numbers].

[42] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. 850–861.

[43] Stanley L Warner. 1965. Randomized response: A survey technique for eliminating evasive answer bias. *J. Amer. Statist. Assoc.* 60, 309 (1965), 63–69.

[44] Samee Zahur and David Evans. 2013. Circuit Structures for Improving Efficiency of Security and Privacy Tools. *IEEE S & P* (2013), 493–507.

[45] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. Cryptology ePrint Archive, Report 2015/1153.