2022

Private Federated Analytics At Scale

Edo Roth
University of Pennsylvania

Private Federated Analytics At Scale

Abstract

Collecting distributed data from millions of individuals for the purpose of analytics is a common scenario – from Apple collecting typed words and emojis to improve its keyboard suggestions, to Google collecting location data to see how busy restaurants and businesses are. This data is often sensitive, and can be overly revealing about the individuals and communities whose data is being analyzed en masse. Differential privacy has become the gold-standard method to give strong individual privacy guarantees while releasing aggregate statistics about sensitive data. However, the process of computing such statistics can itself be a privacy risk. For instance, a simple approach would be to collect all the raw data at a single central entity, which then computes and releases the statistics. This entity then has to be trusted to not abuse the raw data; in practice, it can be difficult to find an entity with the requisite level of trust.

In this thesis, we describe a new approach that uses cryptographic techniques to collect data privately and safely, without placing trust in any party. Although the natural candidates, such as secure multiparty computation (MPC) and fully homomorphic encryption (FHE) do not scale to millions of parties on their own, our key insight is that there are ways to refactor computations in such a way that they can be done using simpler techniques that do scale, such as additively homomorphic encryption. Our solution restructures centralized computations into distributed protocols that can be executed efficiently at scale.

The systems we design based on this approach can support billions of participants and can handle a variety of real queries from the literature, including machine learning tasks, Pregel-style graph queries, and queries over large categorical data. We automate the distributed refactoring so that analysts can write the query as if the data were centralized without understanding how the rewriting works, and we protect against malicious parties who aim to poison or bias the results.

Degree Type

Dissertation

Degree Name

Doctor of Philosophy (PhD)

Graduate Group

Computer and Information Science

First Advisor

Andreas Haeberlen

Keywords

cryptography, privacy, security, systems

Subject Categories

Computer Sciences

PRIVATE FEDERATED ANALYTICS AT SCALE

Edo Roth

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2022

Supervisor of Dissertation

Andreas Haeberlen, Professor of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Committee

Benjamin Pierce, Professor of Computer and Information Science; Committee Chair

Sebastian Angel, Professor of Computer and Information Science, Senior Researcher at Microsoft Research

Aaron Roth, Professor of Computer and Information Science

Srinath Setty, Principal Researcher at Microsoft Research

PRIVATE FEDERATED ANALYTICS AT SCALE

ACKNOWLEDGEMENT

This thesis would not be possible without the endless effort and dedication of my advisor, Andreas Haeberlen. Thank you so much for your support these last five years, and for teaching me so much about the research process, the systems community, and technical writing. Thank you also for withstanding the ups and downs with a smile, and always putting my research development first. I know these last few years have been difficult, but you haven't let it derail our progress and commitment to doing great work.

Of course, this thesis would also not be possible without all the collaborators I've had the fortune of working with during graduate school, and all the contributors to the chapters featured in this work. Thank you to Hengchu Zhang for giving me an example of what a hard-working PhD student should look like (even though I never lived up to that example), and to Daniel Noble for all the time spent on the core archicture of Honeycrisp that forms the backbone of this thesis. Thank you to Karan Newatia, Ke Zhang, Yiping Ma, Elizabeth Margolin, and Tao Luo for all of your hard work and commmitment. Thank you to Sebastian Angel, Benjamin Pierce, and Aaron Roth, for all of your guidance in helping me learn how to do research, as well as for serving on my committee. Thank you to Srinath Setty, Jonathan Lee, and Siddhartha Sen for hosting me at MSR for summer internships and for helping me expand my horizons as a researcher.

Thanks also to the fellow graduate students who've kept me (relatively) sane throughout the years: my fellow Tundra Lab members Neeraj Gandhi, Saeed Abedi, and Robbie Gifford for entertainment and around-the-clock discussion, and my fellow MSR interns Sangeeta Chowdhary and Gurbinder Gill for re-sparking my commitment to research.

I want to thank the friends and family who've gotten me to this point: first, to Mona, for being an excellent companion, and for barking at the mailman on the porch every day to make sure he doesn't steal this thesis. Thanks to the wonderful roommates I've had in Philadelphia over the years: Jacob Arluck, Marc Marin, Kai Mateo, and Lydia

Federico. I am also extremely grateful for the consistent support of Osman Moneer and Justin Woodbridge, who I always know I can rely on. Thanks and love to Bach Tong, my hero, for helping me push through these last couple years.

Finally, thanks to my family: my sisters Ella and Noam, and my parents for their loving support. Nothing is possible without this love. As Naomi Osaka said: "I would like to thank my ancestors, because everytime I remember their blood runs through my veins I am reminded that I cannot lose".

ABSTRACT

PRIVATE FEDERATED ANALYTICS AT SCALE

Edo Roth

Andreas Haeberlen

Collecting distributed data from millions of individuals for the purpose of analytics is a common scenario – from Apple collecting typed words and emojis to improve its keyboard suggestions, to Google collecting location data to see how busy restaurants and businesses are. This data is often sensitive, and can be overly revealing about the individuals and communities whose data is being analyzed en masse. Differential privacy has become the gold-standard method to give strong individual privacy guarantees while releasing aggregate statistics about sensitive data. However, the process of computing such statistics can itself be a privacy risk. For instance, a simple approach would be to collect all the raw data at a single central entity, which then computes and releases the statistics. This entity then has to be trusted to not abuse the raw data; in practice, it can be difficult to find an entity with the requisite level of trust.

In this thesis, we describe a new approach that uses cryptographic techniques to collect data privately and safely, without placing trust in any party. Although the natural candidates, such as secure multiparty computation (MPC) and fully homomorphic encryption (FHE) do not scale to millions of parties on their own, our key insight is that there are ways to refactor computations in such a way that they can be done using simpler techniques that do scale, such as additively homomorphic encryption. Our solution restructures centralized computations into distributed protocols that can be executed efficiently at scale.

The systems we design based on this approach can support billions of participants and can handle a variety of real queries from the literature, including machine learning tasks, Pregel-style graph queries, and queries over large categorical data. We automate the distributed

refactoring so that analysts can write the query as if the data were centralized without understanding how the rewriting works, and we protect against malicious parties who aim to poison or bias the results.

TABLE OF CONTENTS

CHAPTER 1 : Introduction

The analysis of large quantities of data has become ever-present in our modern economy and way of living. Today, personal data is carefully logged, stored, and analyzed by an increasing number of public and private organizations and companies [312]. Often, the data being analyzed can be extremely sensitive and can have significant privacy implications [288]. Balancing the societal value of privacy with the commercial and social benefits of the insights from this "big data" has proven challenging, with countless incidents[1] demonstrating the high stakes of failure by exposing data of hundreds of millions of individuals.

The privacy risk affects data analytics in two different ways. First, where sensitive data is already being processed today, it forces users to trust the entity that is doing the analytics. Analytics are often conducted with weak or no privacy guarantees whatsoever, sometimes relying on ad hoc techniques for anonymity [77, 171, 278, 280]. A prominent example of this is the U.S. Census, which, prior to 2020, relied on a heuristic algorithm of data swapping that has been shown vulnerable to reconstruction attacks that reveal sensitive information about large portions of the population [124]. Second, because of privacy concerns (sometimes coupled with regulation), the analytics that are not performed today lead to missed opportunities for potential insight. The benefits of data analysis are lost when access to rich and well-sourced data (for instance, private health information, or local government statistics) cannot be shared or analyzed at scale [159]. Modern privacy-enhancing technologies give us a way out of both of these dilemmas by shoring up our protection of existing analyses and by unlocking information that was previously thought inaccessible. With the proper tools, the U.S. census can now give provable privacy guarantees in the publicly released statistics for the 2020 decennial census [5, 6]. Instead of locking down sensitive health data, tools now exist to securely analyze this information (in aggregate) without compromising

[1] A small selection of data breach reports: "Report spotlights vast scale of adtech's 'biggest data breach'," https://tinyurl.com/47setecn, "Identity Theft Resource Center's 2021 Annual Data Breach Report Sets New Record for Number of Compromises," https://tinyurl.com/3r3tm97t, "A New Data Breach May Have Exposed Personal Information of Almost Every American Adult," https://tinyurl.com/ydz7jpdk, "Equifax Hack Might Be Worse Than You Think," https://tinyurl.com/y9h4pgsk

any individual's privacy [76]. These technologies can also enable multiple municipal jurisdictions to share data [241] without violating regulations or risking any data compromise. These tools can especially be useful given the huge scale of modern data collection – data analysis on, for instance, over a *billion* iOS devices [18] may lead to tremendous insights!

Federated Analytics is a way to perform analysis on a data set that is distributed across many devices, without collecting the raw data in a central place. Federated analytics can be coupled with strong privacy guarantees for the data holders, such as *differential privacy (DP)* [102]. Differential privacy is a key tool in the toolbox of privacy-enhancing technologies, and has become the gold standard in privacy-preserving statistical analysis. It has been used in the decennial 2020 census [221], as well as by prominent companies such as Google [113], Apple [22], and Microsoft [94]. DP gives formal, provable guarantees on what can be learned about individuals in a database, while still allowing for accurate aggregate data analysis. It's especially useful at scale, or more generally in computing functions with *low sensitivity* – that is, functions that are not impacted greatly by the influence of one individual or data point [230]. Concretely, differential privacy is a property of *randomized algorithms* that release information about sensitive datasets. Algorithms that satisfy differential privacy thus often boil down to computing the exact answer to data queries, and then adding carefully calibrated amounts of random noise to the final result [106, §3.3] (or, sometimes, addingrandom noise to intermediate components before combining them for release [3, 247, 283, 310]).

Differential privacy has seen a wealth of research in the last two decades, and we now understand how to perform many desired analytics tasks and machine learning functions privately. That is, a data analyst can achieve similar performance to the standard non-private algorithms, while gaining an additional bound on the information leakage about any individual. Examples range from histograms [151, 247], to clustering [27, 48], to streaming algorithms [209, 214] and mechanism design [207, 231]. There has also been a range of research on the implementation of differential privacy, ranging from techniques that verify

differentially private analysis [293, 303], to full implementations that manage budgets and answer queries [187, 197, 208], to securing implementations against side channel attacks [144].

However, while DP helps with making the released *output* safe (and resistant to any additional de-anonymizing analysis or attacks), it does not prescribe a methodology to ensure safety of the *computation* itself which leads to this output. In its early literature, which arose out of the database and theoretical computer science community [97, 102, 105], differential privacy was envisioned as being deployed from a central database with a 'trusted curator' who could hold all the data in the clear and carefully dole out access to external data analysts [103]. This single-party, centralized setting assumed that the curator of the database and the data analysts were different parties – however, in practice, we see that this is often not the case. Large companies which aggregate information from their users are often the same companies that benefit from this data analysis to improve their products [248] and create targeted advertising [249]. For the purposes of this thesis, we will refer to the central party which is responsible for collecting *and* analyzing the data as the *aggregator*.

To guarantee that the computation is safe, one naïve approach is to simply assume that the aggregator is one single party who can be trusted with collecting and storing sensitive data, as well privately analyzing it. In practice, however, it can be extremely difficult to find such a trusted entity, given the large spectrum of potential threats for data breaches when data is collected in a single place. A non-trivial percentage of data breaches involve malicious insiders [269], and more than half of the world's population lives in countries where strong end-to-end encryption is illegal without some sort of backdoor or assistance for authorities [189]. These point to the fact that organizations might not wish to have the responsibility of aggregating large amounts of data in the clear, and efforts like Google's FLoC [135] to eliminate cookies for web tracking seem to signal a move towards systems that *never* directly interact with private user data.

There are other approaches which do not assume trust in a centralized party. Namely, in contrast to the 'central' or 'global' model described above, differential privacy research has also expanded widely in the 'local' setting [78, 163]. In this model, the raw data stays with each user on their devices. The noise that is used to achieve differential privacy is added *locally* by each individual to their own data. Once properly perturbed, each data point can be publicly released as differentially private, so users may send their already-noised data to a central party, who aggregates the data, which is now safe to be released and viewed by external parties. This model of local differential privacy, introduced in [102], is known as LDP, as opposed to what we refer to as GDP for the global model. By shifting the onus of enforcing differential privacy to individuals in the LDP setting, there is no longer any need to trust a centralized party – such a party can now safely aggregate the locally noisy values and release differentially private statistics with essentially no additional security infrastructure. Unfortunately, LDP introduces tradeoffs in the accuracy of analytics tasks. Intuitively, because *more noise* is added to each computation (once per data point as opposed to once overall), the analysis is inevitably less precise. Concretely, holding other factors equal, the error for the local model scales with a factor of $\sqrt{n}$, where n is the total number of participants in an analysis, whereas the error for GDP is constant [106, §12.1]. This means that, for instance, an LDP count query involving a million participants could have noise that obscures any signal that is held by less than a thousand participants [113]. In addition, the local model has some security issues as well: [66, 73] show how a small group of colluding nodes can bias the results of an LDP query much more than their numbers would indicate. Because of this, companies like Google seem to be moving away from their earlier forays into local differential privacy [46, 113] in their large-scale implementations.

Additional models lie in between the local and global models – for instance, the shuffle model [112] has in recent years seen a great deal of advancement [29, 74, 130]. It relies on the existence of a shuffling mechanism which can remove any identifying links between the individual participants and their data points that are received by an aggregating entity. By doing so, the accuracy tradeoffs can improve to an error scaling with $\log(n)$ [72]. However,

these guarantees still lag behind the constant error of the global model, and perhaps more significantly, require the existence of more than one trusted party to perform shuffling. To achieve the best possible tradeoff between accuracy guarantees and differential privacy guarantees, the global model remains the highest standard. Therefore, it would be useful to find a way of keeping data safe during computation with an *untrusted aggregator* to have this increased accuracy while also avoiding the assumption of additional parties. Other solutions (e.g., [122]) assume an *anytrust* model where a group of parties is jointly responsible for securing the data – as long as there is one honest party among the set, then privacy guarantees hold. We do not consider this trust setting, and attempt to find solutions where no additional powerful parties are involved outside of a single aggregator.

In this thesis, we present a way to build privacy-preserving federated analytics systems without any trusted aggregator, at a massive scale. Our goal is to achieve the best of both worlds – strong privacy guarantees and high accuracy for a variety of real-world analytics tasks, without placing trust in any single party. The aggregator is responsible for both coordinating data aggregation, and performing the data analysis itself. Throughout the thesis, we assume this is an entity with significant resources, often with access to a data center and large computational power. We assume that there could be a billion participants or more involved in the distributed computation, each holding their own sensitive information on their devices. A summary of the scenario is shown in Figure 1.

1.1. Challenges

Challenge 1: efficient crytography at scale: One challenge with this approach is providing our desired guarantees while supporting the massive scale of a billion participants. Adding additional difficulty, these participants may be geographically scattered, and with far more limited computational resources (e.g., a mobile device or a laptop computer). While modern cryptography has made incredible strides in efficiency and scalability [240], these general techniques are nowhere near ready to be used at such massive scales. Take for instance, secure multi-party computation (MPC) [192, 212, 300], which allows a group of parties to compute a joint function over their respective private data without leaking

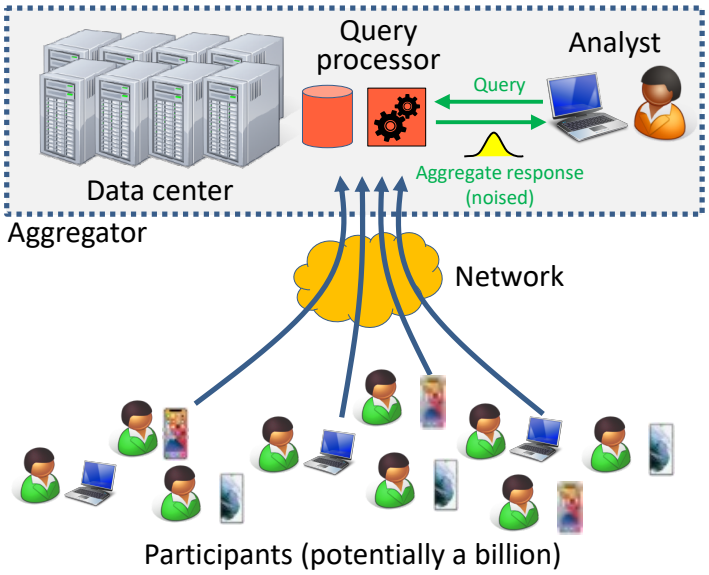


Figure 1: Scenario.

anything about their inputs (beyond what is revealed by the aggregate output). One strawman solution, then, to our problem, is to use one giant MPC – each participant uses their private data as input to a joint function which computes a specified analytics query. The joint function can be specified as differentially private for a specific privacy budget, to protect any leakage from the output. One upside to this approach is that MPC allows us to support very general function specifications. However, while practical MPC efforts have expanded in the last decade [149], they do not scale well in the number of participating parties – current solutions barely support hundreds of parties [292], let alone millions, or even a billion parties! Even significant improvements over the state-of-the-art seem unlikely to support computation at this scale – under current assumptions, even simple functions would take years to compute [23, 174].

Challenge 2: avoiding trusted nodes and handling malicious parties: The second challenge is to perform the computation without having a single (or even a small number of) assumed trustworthy nodes that can become points of failure. Many other solutions (e.g., [79, 122]) take an *anytrust* approach to large-scale private analytics tasks. However,

it may be good to have alternatives – while this setting works if such parties can be found, in practice that isn't necessarily easy to do, especially if a company does not wish to rely on other existing parties to facilitate their data analytics operations.

We also need to be able to handle Byzantine periods of the aggregator. While the aggregator does not necessarily have evil intentions, there can be many temporary issues that the aggregator would want to protect *itself* against (e.g., malicious insiders, government subpoenas, outside hacks and breaches, etc.) In many cases, companies could face massive liability if a breach were to occur [274] – thus, by modeling the aggregator as Byzantine for these periods, we protect against any sort of attack, whether internal or external.

Additional trust challenges come in because of our need to balance accuracy and privacy. One challenge is that malicious users can disrupt the data analytics process, either by colluding with a malicious aggregator in hopes to leak other users' data, or by corrupting their own data inputs in attempts to derail global accuracy. For instance, consider a survey performed by a web browser on how many web pages users open per day. While a typical user may report a number in, say, the hundreds, a malicious user might hack their browser and report a humongous result of 10^{20}. This would completely skew the computation, rendering the results meaninglesss. With a billion users, we can be almost certain that at least a few are going to try attacks of this nature, so there must be defenses in place.

Challenge 3: expressing queries: There are a few challenges here: how to formulate a data query at all, how to prove that it is differentially private, and then how to execute it in a federated setting. First, analysts must be able to reliably write down queries, as it can be difficult for non-experts to correctly write differentially private code that achieves the right tradeoffs between privacy and accuracy. While there are now differentially private programming languages (e.g., [224, 302]), they focus on proving differential privacy, and not on providing a distributed query plan. In the centralized setting, proving differential privacy is the only necessary bottleneck, since executing queries is much more straightforward.

However, in the federated setting, much more is required: queriess must specify actions that need to be performed by each participant to transform their local data, as well as actions that need to performed by the analyst to achieve accurate (and differentially private) results.

Second, even if query is well-formulated, there must also be mechanisms to ensure that differential privacy is reliably enforced, without allowing any room for malicious parties to evade these guarantees. Queries must be transformed from those that are proven to be private in a centralized setting (either manually or automatically), into ones that can be executed in distributed fashion at scale.

Challenge 4: executing queries efficiently: Finally, there is a large body of unique differentially private queries that require suppport. These involve different randomized privacy mechanisms with completely different computational requirements (e.g., the *Laplace* mechanism can typically be evaluated over sums of user data [48, 247, 299], while the *Exponential* mechanism requires evaluating quality scores that often involve comparisons between user data [142, 207, 261]). These different queries also can operate using different parameters (e.g., a small sample of queries using vectors of vastly different lengths and clipping bounds of various sizes [3, 26, 99, 283, 310]).

Our systems must also move beyond data that is structured simply in rows and columns. A common assumption in this scenario is that each participant holds their own data on their devices, making it simpler to aggregate their information by requiring one upload per participant [53]. However, our "individual" data is often highly dependent on others [30], and thus in practice, we may wish to handle non-relational data. This may take the form of structured data such as graphs, which cannot be stored centrally because of privacy. Analyzing graph data brings additional challenges, such as protecting not only each participant's privately held data, but also the data describing their connections and the structure of the graph. Processing graphs at a large scale also introduces efficiency concerns in handling the possibilities of $O(n^2)$ edges for graphs with n participating node

devices.

As we will see, there appears to be no single system that works well for all scenarios, while there are many possible system architectures to choose from. Our challenge is to support as many queries as possible, and to find the best system for any arbitrary query in a massive space of possibilities, where optimizations can be subtle and interdependent.

1.2. Approach

It turns out that there are a few key insights we can make about the nature of queries we would like to have answered, that allow us to construct a more scalable approach than the naïve cryptographic approaches. First, many differentially private queries turn out to have a particular structure that is very helpful. Because of the nature of differentially private noise mechanisms like the Laplace mechanism, these queries are often designed to take advantage of a series of counts and sums, that can then be carefully perturbed to achieve differential privacy. Not all DP queries fall under this umbrella, but our examination of a wide group of categorically different DP queries in published work Orchard [264] found that this structure of sums accounts for an overwhelming majority. Therefore, there do exist efficient solutions at scale – instead of using a framework like MPC which allows for the computation of almost any possible joint function, primitives like *additive homomorphic encryption (AHE)* [257], which scale much better in a federated setting, can be used to support these counts and sums, and thus a wide variety of differentially private queries.

The second insight we have is that for these fairly broad classes of queries that are important in practice, efficient solutions can be found *automatically*, through program transformations and by automatically exploring the design space of possible architectures. This means that the analyst doesn't have to be a cryptographer or a DP expert – they can actually specify the query in a high-level language. Queries expressed in this high-level language (which can be certified as differentially private), can be automatically broken down into the individual components required for distributed execution (computation for different participants in the system, including the aggregator). While there are a wide variety of potential cryptographic

primitives we can make use of, and options for computations to be conducted by different entities, all with complicated interdependencies, we can explore these mechanically using a query planner, instead of forcing a human developer to reason about the best solution for each individual query.

The third insight is that the massive number of users we have in these federated settings is also a key opportunity, because they can help with the cost of the expensive cryptographic operations, which enables queries that the aggregator alone would not be able to handle at this massive scale. This allows us to offload operations like encryption, secure decryption, and verification operations, to the devices themselves. To protect against malicious participants, for instance, we make use of cryptographic techniques like *zero-knowledge proofs* [133], but also use our massive scale as an advantage. Participants are utilized to spot-check the aggregator throughout the computation, and ensure that no malicious behavior results in loss of privacy. While participants are incentivized to make sure that their data remains secure and private, the aggregator is incentivized to make sure that its final results are accurate, so we can balance these needs against each other through cross-checking and verification of each others' actions.

A quick sketch of our solutions is as follows: first, an analyst inputs a query, which is proven differentially private. Each query is transformed into an efficient distributed query plan, where primitives are executed with efficient specialized cryptography (e.g., AHE for sums and counts). Key material will be held by *committees* of user devices, which are elected randomly and accountably from the larger pool of participants, and are responsible for some of the security guarantees of our systems. We rely on the centralized aggregator for certain expensive operations, such as aggregating and storing ciphertexts (which introduces no security risk, since the ciphertexts leak no information about the underlying user data). Some queries (e.g., neural networks [3]) require us to run an aggregation primitive iteratively over many rounds, releasing intermediary differentially private results that can be used for subsequent rounds (e.g., private gradient updates until model convergence). For other

queries (e.g., Pregel-style graph queries [203]), we design entirely new approaches that involve inter-participant communication, routed through the aggregator using a novel semi-centralized mix network to guarantee both data and topology privacy.

1.3. Contributions

Our novel contributions in this thesis are as follows:

- Honeycrisp, previously published as [263], which gives an existence proof for our architectural approach, and specifically shows how to make better use of privacy budget for a recurring count query used by Apple.

- Orchard, previously published as [264], which builds off the Honeycrisp architecture and shows how to support a wide variety of machine learning tasks through query rewriting.

- Mycelium, previously published as [262], which introduces a few additional cryptographic techniques, including a novel onion-routing comunication scheme, to support Pregel-style graph queries.

- Arboretum, which introduces a query planner to support more DP mechanisms, including the exponential mechanism and secrecy of the sample, and automatically finds efficient architectures to execute them.

CHAPTER 2 : Background

2.1. Differential Privacy (DP)

Differential privacy [102] has become the gold standard in privacy-preserving data analysis. It is a mathematical property of randomized functions that take a database as input, and return an aggregate output. Informally, a function is differentially private if changing any single row in the input database results in *almost* no change in the output. If we view each row as consisting of the data of a single individual, this means that any single individual has a statistically negligible effect on the output. This guarantee is quantified in the form of a parameter, ϵ, which corresponds to the amount that the output can vary based on changes to a single row. Formally, for any two databases d_1 and d_2 that differ only in a single row, we say that f is ϵ-*differentially private* if, for any set of outputs R,

$$Pr[f(d_1) \in R] \leq e^{\epsilon} \cdot Pr[f(d_2) \in R]$$

In other words, a change in a single row results in at most a multiplicative change of e^{ϵ} in the probability of any output, or set of outputs.

Differential privacy has another key desirable property, which is that it is resistant to any prior adversary knowledge. For instance, consider the extreme case where an attacker already knows information about $N - 1$ of the N parties in the database. Say that the desired computation is the average over all salaries of individuals in the database, and that Alice is the sole individual in the database whose salary is unknown to the attacker. Even with this prior knowledge, differential privacy still provides protection to Alice, because the randomness present in the computation prevents the attacker from calculating her salary with certainty, giving her plausible deniability, and a concrete bound on the attacker's advantage. This implies that differentially private statistics can be *publicly released* without the worry that they could be combined with any auxilary information to leak more than the guaranteed bound.

For the purposes of this thesis, we introduce a few common differentially private mechanisms. These are all randomized algorithms which satisfy the definition of differential privacy, and can be useful in different contexts.

Laplace Mechanism: The standard method for achieving differential privacy for numeric queries is the *Laplace mechanism* [102], which involves two steps: first calculating the *sensitivity*, s, of the query – which is how much the un-noised output can change based on a change to a single row – and second, adding noise drawn from a Laplace distribution with scale parameter s/ϵ; this results in ϵ-differential privacy. Differential privacy is also compositional, that is, if we evaluate two functions f_1 and f_2 that are ϵ_1- and ϵ_2-differentially private, respectively, publishing the results from both functions is at most $(\epsilon_1+\epsilon_2)$-differentially private [107]. This property is often used to keep track of the amount of private information that has already been released: we can define a *privacy budget* ϵ_{max} that corresponds to the maximum loss of privacy that the database participants are willing to accept, and then deduct the "cost" of each subsequent query from this budget until it is exhausted.

Exponential Mechanism: The *exponential mechanism* [207] is a common choice for working with categorical data, or data for which ordering of distinct data points is crucial (e.g., auctions). Suppose each user's data is from some discrete range R (the possible "categories" to which the user can belong), and we have a quality score $q(r, d)$ that defines, for each $r \in R$, how "useful" output r would be if the database were d. If the quality score q is Δ-sensitive in its database argument, the exponential mechanism then outputs each possible $r \in R$ with probability proportional to $e^{\epsilon q(d,r)/(2\Delta)}$; this once again results in $(\varepsilon, 0)$-differential privacy. An alternative but equivalent implementation is to compute $q'(d, r) := q(d, r) + \mathit{Gumbel}(2\Delta/\varepsilon)$ and to then return $f'(d) := \text{argmax}_r\ q(d, r)$ [106, §3.4]. When releasing multiple outputs (e.g., the k most frequent items), one can either draw Gumbel noise k times to maintain $(\epsilon, 0)$-differential privacy or simply add noise once and release the outputs with the k highest scores, which yields $(\sqrt{k} \cdot \epsilon, 0)$-differential privacy [99]. For this work, we also sometimes refer to (ε, δ)-differential privacy. A common recommendation is

that δ should be at least smaller than $1/N$, where N is the total size of the [167].

Sampling: Another useful primitive is *secrecy of the sample* [276]. Let $\sigma(d, \phi)$ be a function that selects each element of some database d with probability ϕ, and suppose we have a query $f(d)$ that is $(\epsilon, 0)$-differentially private. Then $f(\sigma(d, \phi))$ is $(ln(1 + \phi(e^{\epsilon} - 1)), 0)$-differentially private, *as long as* nobody can observe which elements have been selected. When $\epsilon \leq 1$ and ϕ is sufficiently small, this is close to $(\frac{2\phi}{\epsilon}, 0)$-differentially private. Notice that this multiplicative bound also can improve δ - if we have a query $f(d)$ that is (ϵ, δ)-differentially private, then $f(\sigma(d, \phi))$ becomes $(ln(1+\phi(e^{\epsilon}-1)), \phi \cdot \delta)$-differentially private.

Composition Theorems: By default, differential privacy composes linearly in the privacy parameter ϵ – that is, suppose that two algorithms A and B are ϵ_1 and ϵ_2-differentially private, respectively. If C is the algorithm which combines both A and B (releasing a pair of outputs), then C is $\epsilon_1 + \epsilon_2$-differentially private [102]. However, there are other, more advanced composition theorems [106, §3.5] that introduce tighter bounds dependent on the properties of the databases and computations [101, 107, 166, 295]. This includes mechanisms that allow for adaptively choosing queries and reasoning tightly about the worst-case privacy loss, including the *sparse vector technique* [106, §3.6].

2.2. Secure Multi-Party Computation (MPC)

Secure multi-party computation (MPC) protocols allow a set of (potentially distrustful) parties to securely compute functions over their joint inputs in such a way that the protocol executions reveal nothing about these inputs, except what is already implied by the output of the computation [192].

MPC has been researched extensively by cryptographers and security researchers for several decades, since its introduction in the 1980s by Yao for the two-party case [300], and by Goldreich, Micali and Wigderson for the multiparty case [212].

In recent years, MPC has become efficient enough [23, 174, 256, 292] to be used in practice, and has made the transition from an object of theoretical study to a technology being

deployed in industry, with use cases ranging from examining gender and racial wage gaps in the city of Boston [183], to financial oversight [4, 49], medical computations [76], and satellite collision detection [152]. There are currently several general-purpose MPC implementations suitable for practical use – a good summary of the current state-of-the-art, as well as their limitations and challenges going forward can be found at [149]. These general-purpose MPC frameworks reduce the burden of designing custom cryptographic protocols, and are intended for use by non-experts in cryptography. These frameworks differ in a few different ways, including the threat model they defend against (semi-honest adversaries [50, 250] vs. malicious adversaries [168, 176, 291]), and the number of parties they support (two parties [291], three parties [50] and multiple parties, up to at least a few dozen given today's state of the art [168, 176]).

2.3. Homomorphic Encryption (HE)

Homomorphic encryption is a family of encryption schemes that allows any data to remain encrypted while certain operations are performed on it [7]. These operations are preserved across the plaintext - for instance, adding two ciphertexts can produce a ciphertext which is the sum of the two underlying plaintexts:

$$Enc(X + Y) = Enc(X) + Enc(Y)$$

This can be incredibly useful to perform computations on top of data that will *never be seen in the clear*! Generally, homomorphic cryptosystems are like other forms of public key encryption [257, 258], in that they use a public key to encrypt data, and allow only the party with the matching private key to access its unencrypted data (though there are also examples of symmetric key homomorphic encryption [93], and generic methods of converting between the two kinds of schemes [265]). This means that Alice can generate a keypair, encrypt her secret information in a series of ciphertexts, and send it to a third party to perform some computation over her encrypted data, with no risk of her secrets being revealed. Once the computation has been completed, Alice can decrypt the single ciphertext containing the results of her desired computation.

There are a few different classes of homomorphic encryption:

- *Partially homomorphic* schemes support only one type of arithmetic operation, e.g., addition or multiplication [41, 109, 131, 234].
- *Somewhat homomorphic* encryption schemes can evaluate two types of operations, but only for a subset of circuits [55, 267].
- *Leveled homomorphic* encryption schemes support multiple kinds of operations, but only for circuits composed of gates of bounded depth [137].
- *Fully homomorphic encryption (FHE)* support the evaluation of arbitrary circuits composed of multiple types of gates of *unbounded* depth [57, 114, 125].

Of course, fully homomorphic encryption (FHE) is the most powerful form of HE, supporting both addition and multiplication over encrypted data. In combination, this allows for arbitrary computation. While it was, in past decades, unclear if FHE was impossible or impractical, advances in the last decade in particular [58, 126, 147, 275] have made this realizable and practical for real computations.

The first construction for an FHE scheme was proposed by Gentry [125], and makes use of lattice-based cryptography. In the next few years, additional schemes were proposed, including the BFV [114] and BGV [57] schemes – these gain their security based on the hardness of the (Ring) Learning With Errors (RLWE) problem [200]. There are now many additional FHE schemes which continue to improve their efficiency and support expanded use cases. The CKKS scheme [71], in particular, is useful for encrypted machine learning because it encrypts approximate values as opposed to exact values, and deals efficiently with the noise resulting from machine learning.

There are several FHE libraries in production today, perhaps most notably the open source

PALISADE [1] developed by a consortium of institutions, and Microsoft's SEAL [2]. There are even compilers that optimize for homomorphic encryption, for instance CHET [87], which is designed for tensor programs that allow neural network inference over encrypted data. There is an open consortium which meets to standardize the parameters of homomorphic encryption schemes and guarantee their security [14].

2.4. Zero-Knowledge Proofs (ZKP)

A zero-knowledge proof (ZKP) is a cryptographic method by which one party (designated as "the prover") can prove to another party ("the verifier") that a given statement is true, without revealing *any* other information that cannot be inferred from the statement itself. Introduced by Goldwasser, Micali, and Rackoff in the 1980's [133], all ZKP's must satisfy the following three properties:

- **Completeness**: If a statement is true, and the the prover and verifier both follow the protocol correctly, then the verifier will accept the proof.
- **Soundness**: If the statement is false, the verifier will not be convinced by the proof.
- **Zero-Knowledge**: If the prover follows the protocol, and the statement is true. then the verifier will be convinced by the proof, without learning any information from their interaction.

ZKP's have useful applications where computations are executed remotely over private data (and have picked up steam in particular in the blockchain and cryptocurrency spaces, e.g., in the ZCash [154] cryptocurrency based on the Zerocash protocol [268]). A particularly useful class of ZKP's are Succinct Non-Interactive Arguments of Knowledge (SNARKs or zk-SNARKs [45]). These "non-interactive" schemes, unlike previous instantiations of ZKP's, do not require multiple rounds of interaction between the prover and the verifier. Rather, they simply require the prover to send a short cryptographic proof, which can be verified

[1] Palisade Homomorphic Encryption Software Library: An Open-Source Lattice Crypto Software Library. https://palisade-crypto.org/

[2] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL

efficiently by the verifier, in time typically much less than is required to generate the proof itself [238]. This is helpful in offloading computation to third parties, because the correctness of operations can be provably shown using far fewer resources than the computation itself requires. "Succinct" zero-knowledge proofs can often be verified within a few milliseconds, with proof lengths of only a few hundred bytes, even for statements about computations whose circuit representations are large [61].

There are many existing zero-knowledge proof systems today with different properties and tradeoffs, e.g., [16, 39, 61, 81, 204, 238, 290]. While some schemes require a trusted setup to generate randomness that can be used for the protocol [81, 238], *transparent* protocols (referred to as zk-STARKs [38]) do not require this trusted setup. However, the proofs are several orders of magnitude larger than those of zk-SNARK schemes [37, 38]. Most state of the art systems have prover complexity that scales either linearly or nearly-linearly in computation size [62, 75], with verifying time being close to constant (with the exception of a linear dependency on public inputs)[140][3]. Different proof systems also optimize for different kinds of computations (e.g., memory efficiency for larger circuits [294], or circuits that can be composed [172]).

[3]See Section 5.4 for concrete costs: https://docs.zkproof.org/pages/standards/accepted-workshop4/proposal-aggregation.pdf

Other privacy guarantees

Other definitions of privacy (besides differential privacy) have been proposed in the past. For example, k-anonymity [279], which requires that in a database containing values for multiple attributes, every possible combination of identifying attributes must occcur in at least k different rows of the data set, to prevent identifying any single individual (for a tunable parameter k). Follow-up definitions and variants include notions like l-diversity [201], t-closeness [190], and n-confusion [277]. However, in light of large-scale de-anonymization attacks such as [223], the research community has recently focused more on the more robust notion of differential privacy, which is a property of the data analysis *process*, not the *dataset* itself (and protects against adversaries with any external knowledge).

There are also cryptographic notions of secrecy, with security against both computationally-bounded adversaries (i.e., semantic security [131]) and information-theoretic adversaries (i.e., unconditional or perfect secrecy [271]). These definitions are both much stronger than differential privacy, and are not suitable for our desired use case, where we still want to extract some meaningful information out of the data. In fact, these definitions are predicated on proving that no adversary can learn any non-negligible amount of information from their interactions.

Many other systems for collecting sensitive data rely on secret-sharing [8, 161, 177], anonymizing networks [118, 186] or Tor-liike systems [153, 245] to aggregate the data privately.

Some solutions, such as [191] use trusted hardware like Intel's SGX. We avoid this approach because current TEE implementations are not yet sufficiently trustworthy, as shown, e.g., by the many successful attacks on SGX [228].

Local or Shuffle Model

Several distributed differentially private systems add noise locally to each user's input, instead of once to the final result. This avoids the need for expensive cryptography, but it requires more noise, and thus reduces accuracy. One prominent example of such a system is Google's RAPPOR [113, 115].

Similar systems have been deployed, e.g., by Apple [22], Microsoft [94], and Snap [242]. As discussed in Chapter 1, Local differential privacy (LDP) requires significantly more noise than global differential privacy (GDP), which can be limiting in practice [46], and it is vulnerable to attacks from small groups of colluding users [66, 73].

The schemes of [8, 138] also require participants to add noise locally, however, rather than use homomorphic encryption to hide the users' inputs from the aggregate, they use pairwise blinding factors. Additional theoretical contributions have also operated in the local setting, but have included additional cryptographic tools [67, 164, 272].

Prochlo [46] additionally introduced an Encode-Shuffle-Analyze (ESA) architecture. While in the original paper, they claim to only provide LDP, the guarantees that Prochlo provides have been re-analyzed in [111] after theoretical revelations on the amplification of privacy that shuffling can give ([29, 74, 112]). This architecture generally relies on the existence of additional parties to perform the shuffling that breaks anonymity links between participating data contributors and an aggregator.

Trusted parties, multiple servers or anytrust (diff. trust assumptions)

Some existing systems rely on a trusted party – an assumption that our approach avoids. For instance, the aforementioned Prochlo requires a shuffler, PDDP [69] makes use of a proxy to send information from a client to an analyst, [84] uses a trusted third party, and [188] relies on a trusted dealer to set up keys. [251] does not rely on trust, but operates in a different setting where users communicate directly with each other.

Another group of prior solutions relies on the *anytrust* model, that is, a *group* of third

parties that must include at least one honest party in order to protect privacy. These parties are static, which could potentially make them known targets for an adversary, and each of these parties must contribute substantial resources. One example of such a system is UnLynx [122], which uses a group of trusted servers to help with shuffling, aggregation, and query processing. UnLynx supports rich queries(e.g., a SQL-style `GROUPBY`), but the servers' workload grows linearly with the data size, so, with a billion users, each server would have to be quite powerful. Cryptϵ [266] supports GDP without a trusted party, but requires two non-colluding semi-honest servers. Their approach involves encrypted data so that the aggregator never sees the data in the clear, and rely on a 'cryptographic server' to maintain the keys for decryption. If the cryptographic server were to be compromised (or collude with the aggregator,) privacy would be lost.

Prio [79] is another example from this group that also relies on a group of special servers for aggregation. As with UnLynx, each server needs substantial CPU and bandwidth resources. Like [209], Prio does not provide differential privacy; rather, it focuses on robustness to malicious user inputs, which it recognizes using a novel kind of zero-knowledge proof. This makes Prio vulnerable, e.g., to intersection attacks, in which an analyst performs two identical queries but forces one device to be offline during the second query, so that its sensitive data can be computed from the two results. [79] does however, sketch a possible extension to add differential privacy, and is currently being deployed at Mozilla.

Smaller Scale

A variety of solutions are available for systems with at most a few thousand users. For instance, Shi et al.[272] use a distributed key generation scheme to remove trust in the aggregator, and [8] use pairwise blinding to avoid expensive encryption (pairing up users and adding complementary amounts of randomness to hide each others' inputs). These approaches face some challenges under churn, since they require devices to be online for security guarantees.

Some systems have scaled MPC to impressive sizes – for instance, SEPIA [63] handles hundreds of users, and Reyzin et al. [255] perform secure aggregation for thousands, by adding homomorphic threshold encryption – but supporting millions of users with MPC seems unrealistic. Bonawitz et al. [53] use secret sharing, but, with n users, several costs grow with $O(n^2)$; Bindschaedler et al. [44] and Goryczka and Xiong [138] require $O(n^2)$ communication; Rastogi and Nath [251] use (t, n)-threshold encryption; and Halevi et al. [145] have $O(n)$ latency, since users must interact with the aggregator sequentially.

In particular, Bindschaedler et al. [44] considers differentially private aggregation with an untrusted aggregator and a strict star topology (users never communicate with each other). Their system uses both local and global noise addition to provide security against collusion between the aggregator and users. However, they require each participant to perform $O(n)$ public-key encryptions, and require $O(n)$ communication between each user and the aggregator. This may not be suitable in our setting, where n is very large, possibly up to a billion.

Halevi et al. [145] shows how to compute an *arbitrary* function in a setting where there are n users and single server. [145] does not guarantee differential privacy, but the full version of the paper does outline a system for securely computing a sum in this model [146][§ 4.3]. However, the users connect to the server *sequentially*, and each interaction with the server requires $\Omega(n)$ communication, which limits scalability.

Secure Aggregation

Summation is a key building block in this solution space, and there are solutions that focus on this secure summation, ensuring that no individual data point is revealed until the complete sum has been calculated. Bonawitz et al. [54] considers such a protocol in a scenario that is similar to ours, and presents a protocol that also offers strong protection against user drop-out during protocol runs. It uses pair-wise blinding to hide user inputs (as in [8, 138]), but does not focus on differential privacy. Instead, the server learns the *exact*

summation, but only if a certain threshold of inputs are received. This approach requires pair-wise key exchange between all parties (and thus $\Omega(n^2)$ communication); scalability is achieved by performing the aggregation in many small batches of n values (in the evaluation, $n \leq 500$). Since the threshold is less than n, the anonymity set is on the order of hundreds, even if there are millions of users.

[35] updates the previous work from Bonawitz et al., and shows how to achieve poly-logarithmic overhead in this single-server secure aggregation setting, which allows them to scale up to a billion users. They also show how this architecture can support DP in the shuffle model (see above).

Federated Learning

Federated Learning (FL) [42, 52] is another approach to working with highly distributed data. The setting is similar to ours, with one central aggregator and many distributed participants. The aggregator maintains a model through a set of parameters, and this model is iteratively distributed to clients, who update the model through training on their local data. The model eventually converges to a joint representative model over all the clients' data. While FL has been proposed as a privacy-centric model for large-scale machine learning because no raw data leaves the devices, in practice the model updates (and the final model itself) can leak significant information about the underlying data[210, 273]. Most existing systems do not guarantee differential privacy, and the ones that do typically rely on LDP, such as [3]. The work of [127] shows a promising algorithmic approach to central differential privacy using random sub-sampling and central distortion, but they do not have any systems contributions to guarantee privacy, and they rely on a trusted curator.

Zhu at al. [310] recently proposed an interactive protocol with better privacy, specifically for discovering heavy hitters, but it does trust the aggregator with one simple task (thresholding). Truex et al. [285] relies on threshold Paillier, but it is limited to small deployments. Oort [181] carefully selects clients for participation to achieve better ML guarantees while

improving system efficiency. It does so by prioritizing clients who either have data that offer the greatest utility in improving model accuracy, or have capabilities of quick training. However, they do not guarantee DP (although they do optionally guarantee LDP).

[165] gives an overview on the current state of FL, and the remaining challenges in achieving privacy ([165, §4]) and efficiency at scale in practical settings ([165, §3, 7]).

CHAPTER 4 : Honeycrisp

The first step is to show, as an existence proof of sorts, how to design a system that answers even *a single* query efficiently and privately at scale. We focus on a query inspired by Apple's deployment [19] of differentially private analytics.

The implementation of this query falls into a common pattern of answering queries about a data set in a differentially private way: 1) computing the exact answer to the query, and to then 2) adding a carefully calibrated amount of noise to the answer before returning it to the client.

For concreteness, let us consider one specific use case from Apple's deployment in a bit more detail. To get a better sense of how popular each emoji is, Apple devices record an event every time the user types an emoji – assuming the user has opted in – and temporarily store the events, with appropriate noise added in, locally on the device. Then, once in a while, the device samples a subset of these events and sends them to Apple's servers [22], where they are aggregated with events from other devices and then analyzed.

Existing deployments like the one described above face two important challenges. The first has to do with the way the data is collected.

In Apple's deployment, noise is added locally by each user *before* the contributions are collected by Apple. Adding noise locally, before aggregation, is necessary for user privacy, since otherwise Apple would have access to the user data in the clear and could be compelled to collect and reveal the data of individual users. (Apple receives thousands of requests for data from law enforcement every year [20].)

Although LDP is clearly better for privacy, it also adds considerably more noise to the overall data set and thus reduces the accuracy that can be achieved from comparable queries. The differential privacy literature reasons about this tradeoff between privacy and utility by assuming a *privacy budget* that reflects the users' privacy expectations; it then assigns a

"cost" to each query that reflects the amount of information the query can leak and that must be deducted from the budget each time the query is asked. In general, LDP requires a much larger privacy budget than GDP because, to achieve similarly accurate results, the amount of perturbation of each data point must be significantly lower.

The second challenge has to do with the fact that new data is uploaded regularly (e.g., daily). Regular updates are necessary because user behavior can change over time and Apple or Google would presumably like to track such changes; however, it also means that, even if the answers are appropriately noised, the noise terms from repeated queries will eventually cancel out as more and more queries are answered, revealing statistics about the user's behavior. This leakage further exacerbates the first problem: to get the same level of accuracy, the privacy budget would need to be even larger! If one stops answering queries once the budget is exhausted, this approach provides strong guarantees. However, no finite budget would be enough to support periodic queries indefinitely, which is why Apple opted to replenish the budget *every day* [282]. This would be reasonable if 1) users were comfortable with potentially revealing emojis they typed yesterday, or 2) the emoji usage by the same user on two different days were completely uncorrelated; however, neither seems like a realistic assumption.

In this chapter, we propose a possible way out of this dilemma. We present a system called Honeycrisp that can sustainably run queries like the one from Apple's deployment while protecting user privacy *in the long run*, as long as the underlying data does not change too often – which is likely, e.g., in the case of emoji usage patterns. Honeycrisp accomplishes this with a combination of two key insights. The first is a new threat model, which we call *occasionally Byzantine + mostly correct (OB+MC)*, and which we have specially tailored to large-scale deployments with millions of users, such as Apple's or Google's. In contrast to prior work, such as Prochlo [46], UnLynx [122], or Cryptϵ [266], we do not assume powerful third parties that could take on a substantial amount of work: with millions of users, any substantial involvement would require a lot of resources – perhaps even a data center, which

few parties can afford. On the other hand, we assume that the adversary can compromise at most a small fraction (say, 1–5%) of the users' devices. This is substantially lower than the usual 1/2 or 1/3, but, at the scale we are targeting, it would still mean far more corrupted devices than are found, e.g., in a typical botnet.

Our second insight is that, in this model, we can use a cocktail of cryptographic techniques – specifically, multi-party computation (MPC) and a form of homomorphic encryption – to efficiently implement global differential privacy *at scale*, which enables us to leverage the sparse vector technique (SVT) [104, 261] from the differential privacy literature. We introduce a technique we call *collect-and-test (CaT)* that can accomplish this, and we present a concrete set of algorithms that implement CaT, along with a security proof. Interestingly, our approach *does not require a trusted party at all.* Even the system operator itself (e.g., Apple or Google) does not need to be trusted; Honeycrisp uses it only to facilitate the computation by providing resources, such as computation and bandwidth.

Using a prototype implementation, we demonstrate that Honeycrisp can support a form of aggregation that is common in both Apple's and Google's current deployments and would be fast enough to run at scale, with billions of user devices. With a billion devices and our choices for the cryptographic building blocks, the aggregator would need to provide roughly 1.2 MB of bandwidth per user per query, and less than 50 cores; most user devices would need to provide about 1.2 MB of traffic and about 60 seconds of computation time, although a tiny, randomly chosen set of devices would need to provide substantially more. We also show that, with comparable security and privacy, a LDP system would exhaust a typical privacy budget after only 91 days, whereas Honeycrisp could run for up to ten years. In summary, our contributions are:

- the collect-and-test technique (Section 4.1);
- the design of Honeycrisp (Section 4.3);
- a prototype implementation (Section 4.5); and
- an experimental evaluation (Section 4.6).

4.1. Overview

Our scenario is the same as described in Chapter 1, there is a very large number of users (e.g., all iPhone and MacBook owners, or all Chrome users), as well as one central *aggregator* $\mathcal{A}$ – e.g., Apple or Google. Each user regularly collects some sensitive information on her device that she wishes to make available to the aggregator for analysis, provided that her privacy can be guaranteed. The aggregator has substantial computational resources – e.g., a data center – and is able to collect the uploaded data from the devices, as well as perform some cryptographic operations. The aggregator also has at least one *analyst*, who would like to issue queries about the collected data; for instance, one possible query could be a count-mean sketch of emojis or new words that are not yet in a dictionary, as in [22].

4.1.1 The OB+MC threat model

To provide strong protections, we would like to be robust not just to honest-but-curious (HbC) behavior, but rather to actual Byzantine faults. At smaller scales, the standard threat model for this setting would be to assume that a certain fraction (usually one third) of all nodes can be Byzantine. However, this seems overly pessimistic for our scenario, for two reasons.

Aggregator: Occasionally Byzantine (OB). First, the enormous size and prominent position of the aggregator would subject it to a lot of scrutiny (from the press, the users, etc.), so it is not likely to be Byzantine for long. It can very well be Byzantine for brief periods, however – for instance, due to misbehavior by rogue employees. Because of this, even a well-intentioned aggregator might not "trust itself" to always behave correctly, and might wish to design the system to limit the damage it could do during any Byzantine periods.

Users: Mostly Correct (MC). Second, if the number of users is very large (e.g., the 1.3 billion macOS/iOS devices [18]), it seems unlikely that an adversary could compromise a large fraction of them. This is different from, say, BFT: in a replica set of 4–7 nodes,

compromising 1/3 of the system means just one or two nodes. But at the scale of the Apple ecosystem, even compromising 3% would mean about 39 *million* nodes, which is much larger than, e.g., a typical botnet.

We refer to these assumptions as the *OB+MC* threat model, to distinguish it from the classic Byzantine fault model and its 1/3 failure threshold. To reiterate, we assume that a) the aggregator is HbC when the system starts and usually remains HbC, except for occasional periods of Byzantine behavior, and that b) a *small* fraction of the devices, on the order of a few percent, is Byzantine as well. Notice that the latter requires that the aggregator refrain from building back doors into its devices, so that, during its Byzantine periods, it cannot – or will not [21] – change the devices' software.

We explicitly do *not* assume the existence of a trusted third party that is willing to be actively involved. If a party is available that can be trusted with some very limited tasks, such as generating random bits, Honeycrisp can use it for efficiency (as described in Appendix B, but it is not required.

Goals: Our primary goal is to protect user privacy. When the aggregator is behaving correctly, we also ensure integrity (that is, accurate query results), but we drop this second goal during the aggregator's Byzantine periods. This seems reasonable, since the aggregator is the beneficiary of the collected data and can only harm itself by misbehaving.

4.1.2 Background: The Sparse-Vector Technique

A standard method for achieving differential privacy for numeric queries, which we hope to take advantage of in this context, is the Laplace mechanism. The Laplace mechanism, in combination with a finite privacy budget, cannot support repeated queries indefinitely, since the budget will eventually be exhausted. However, the following, different mechanism allows an analyst to make regular, repeated queries without significantly reducing the privacy budget with each query. The analyst does not ask for $f(x)$ directly; instead, she provides a "guess" $\hat{f}$ and asks only whether $|f(x) - \hat{f}| > \hat{T}$, where $\hat{T}$ is some small, noised threshold.

The actual value $f(x)$ is then released *only* if the answer is yes. This is called the *sparse-vector technique (SVT)* [104, 106, 261].

The SVT has the key advantage that the privacy budget needs to be charged significantly only if the answer to the threshold query is yes – that is, if the answer to the query does differ from the analyst's guess. (Intuitively, the reason is that the analyst does not really learn anything new if the guess was correct.) A small charge is necessary even if the answer is no, but, via advanced composition [107], this charge can be "prepaid" at the beginning and amortized over a large number of queries. Thus, the privacy budget is depleted mostly in proportion to how frequently the data *changes*, with an additional logarithmic decay to account for negative answers and the possibility of error in threshold comparison. The details for this privacy budget improvement are discussed in detail in Section 4.6.2. In our motivating scenario, such changes (different emoji preferences, or appearance of new, previously unknown words) are likely to be rare. Thus, the SVT enables the analyst to run the system for much longer, or even indefinitely, without assuming that the users are willing to tolerate high worst-case information leaks.

4.1.3 Strawman solutions

Collect the data unencrypted: One way to implement the SVT would be to simply have the aggregator collect all the data unencrypted, and to perform the thresholding at the aggregator. In our threat model, this is not an option: the aggregator could become Byzantine at any time and would then be able to leak the plain-text information of any user.

Use large-scale MPC: Another way would be to implement the SVT using a multi-party computation (MPC) between all the devices. Each device could input its local data, and the MPC could then aggregate the data, do the thresholding, and then either release the new answer or indicate that the answer has not changed. However, generic MPC is known to scale very poorly with the number of participants: efficient MPC techniques are available for two parties (e.g., [175]) and some can handle dozens of parties (e.g., [292]) but we are

not aware of any technique that could be used for a billion parties.

Use small-scale MPC: The MPC could also be performed at a smaller scale, e.g., between the aggregator and one device, or a small subgroup of devices. However, this is risky because we have assumed that the aggregator is capable of small-scale collusion and/or a small-scale Sybil attack – for instance, they could manufacture a few extra devices, keep them, and always perform the MPC with these devices. Also, it is not clear how the data would be aggregated: individual devices (e.g., phones and tablets) are not likely to be capable of receiving and processing millions of records from other users, nor can they necessarily be trusted with this information.

4.1.4 Our approach: Collect-and-Test

We now sketch our actual approach, which we call *collect-and-test (CaT)*. CaT proceeds in the following three phases.

Setup phase: In the first phase, CaT uses a sortition scheme (Section 4.3.2) to randomly and accountably choose a *committee*, which is a small subset of devices. The committee then uses MPC to generate a keypair for an additively homomorphic cryptosystem. The private key is secret-shared, and the shares are kept on the committee's devices, whereas the public key is endorsed by the devices and sent to the aggregator (Section 4.3.3).

Collect phase: In the next phase, the aggregator uses its resources to distribute the public key and the endorsements to all the devices; each device verifies the endorsements (Section 4.3.4), then encrypts her data with this key, and sends the ciphertext back to the aggregator, along with a zero-knowledge proof that the encrypted plaintext is formatted correctly and in the right range. (Note that the aggregator does *not* know the private key for the cryptosystem and thus cannot perform these checks on the plaintext directly!) Finally, the aggregator verifies the range proofs, aggregates the ciphertexts using the homomorphic property of the cryptosystem, and thus obtains a single ciphertext that contains the (precise, un-noised) sum of the individual records (Section 4.3.5).

Test phase: Finally, the aggregator sends the (single) aggregate ciphertext back to the committee, along with its "guess" for the plaintext value. The committee members input their key shares, the guess and the ciphertext into another MPC, which combines the shares, recovers the private key (Section 4.3.6), and decrypts the ciphertext to obtain the precise sum. The MPC then generates random bits to noise the sum (Section 4.3.7) and compares the result to the aggregator's "guess" (Section 4.3.8). If the difference is larger than the threshold, the MPC outputs the true result; otherwise it outputs a default value to indicate that the result is close to the guess.

4.2. Challenges

At first glance, it may seem that the key ideas are only in the approach (e.g., the applicability of the SVT and homomorphic encryption), and that an implementation of CaT could simply consist of a few standard cryptographic building blocks. However, there are also two subtle technical challenges. First, although the aggregator cannot directly read the encrypted data, it can attempt to infer the data in other ways – e.g., by leaving out some ciphertexts while computing the aggregation, and/or by fabricating Sybil identities that will adaptively choose the ciphertexts they contribute (for instance, identical to the ciphertext of a specific user whose data the aggregator wants to learn). To address this, we have developed a verifiable aggregation protocol for the Collect phase that can ensure that the aggregator 1) includes the ciphertext of each user exactly once, 2) computes the aggregation correctly, and 3) can include at most a small fraction of malicious (but non-adaptive) inputs. The second challenge is scalability: with easily a billion participants that each have only very limited resources, we must design the protocol very carefully to avoid overwhelming individual participants.

4.3. The Honeycrisp system

Next, we describe a concrete system called Honeycrisp that implements the CaT approach in the OB+MC model. Honeycrisp relies on the following assumptions:

1. Each device i has a locally generated keypair σ_i/π_i for signing messages; the aggregator can check whether each public key π_i belongs to a valid device.

2. There is a once-off randomness beacon – an independent party P that can be trusted to generate a single random string, B_0, when the system is first launched.
3. All devices know an upper bound N_{max} and a lower bound N_{min} of the number of potential participating devices in the system.
4. There is an immutable bulletin board B that the aggregator can use to broadcast a small amount of data to all devices.
5. Devices can use an external, time-stamped channel X to report the aggregator if it behaves maliciously.
6. Secure, authenticated, point-to-point channels can be established from each device to a) the aggregator, and b) a small number of other devices.
7. There is an upper bound f ($\approx$ 1–5%) on the fraction of participating devices that may be malicious, collude with each other, or collude with the aggregator.
8. There is an upper bound g on the probability that an honest device goes offline while participating in a round.
9. There exists an efficient hash function that is indistinguishable from a random oracle.

For instance, in the case of Apple, these assumptions could be satisfied by 1) the Secure Enclave coprocessor in recent devices; 2) an existing random number service, such as `random.org`, or a widely respected party, such as the EFF; and 3) public estimates on the number of devices sold [219], and/or self-reported statistics on installed base of iPhone users [296]; again, only an imprecise range is necessary. 4) could be any of several (free, centralized) "bulletin boards", such as Wikipedia, StackExchange, or Reddit; only the aggregator needs to post transactions, and only a small number of times per round, so neither cost nor latency should be an issue. For 5), if users have evidence that the aggregator has acted maliciously, they could post this evidence in an online forum (Twitter, Wikipedia, ...) or give it to the press. 6) could be satisfied with TLS channels, in combination with NAT traversal techniques [116]; 7) seems plausible given experience with existing deployments (see 4.1.1); 8) seems plausible given the always-on nature of modern devices (which is being

leveraged, e.g., for push notifications), and 9) is a common model for cryptographic protocols. For additional details about our assumptions and ways to satisfy them, please see Appendix C.1.

We also make a simplifying assumption, which is that most users have only one device, and that it is therefore sufficient to provide a per-device privacy guarantee. However, Honeycrisp can easily be changed to give a per-*user* privacy guarantee instead – by selecting a single device for each user (e.g., based on AppleID) and by having only this device respond to queries, using data from that user's entire set of devices.

4.3.1 Committees and rounds

Recall from Section 4.1.4 that there is a committee of C devices that holds the shares of the private key for the homomorphic encryption, and that also maintains the privacy budget. Since the committee is composed of regular devices, it would be very burdensome to require the same devices always perform the role of the committee. Hence, Honeycrisp segments its execution into discrete *rounds*, and it randomly appoints a new committee for each round.

The security of the scheme is contingent on the dependability of this committee. Since we cannot trust individual devices, any action that could cause sensitive data to leak ("privacy failure"), such as making decisions on behalf of the committee or reconstructing the secret key, must require a large subset of, say, A members. But A cannot be too large either, otherwise it can happen that some queries do not receive an answer ("liveness failure") because some committee members – say, B devices – go offline during a round.

In our design, we chose $A = \frac{2}{5}C$ and $B = \frac{1}{5}C$. Using a probabilistic argument, we can show that, if up to $f = 3\%$ of the devices are malicious and the system runs one round per day for ten years, $C \geq 29$ is sufficient to prevent privacy failures with probability 99.999%, while ensuring that at least 95% of the queries are answered successfully (with an unsuccessful query simply resulting in a re-run in the subsequent round). We provide more details in Section 4.6.4 and the full analysis in Appendix C.3.

4.3.2 Setup phase: Sortition

Next, we show how the committee can be selected in such a way that an adversary cannot influence or predict the selection. This particular building block has appeared in several earlier systems, including Algorand [128] and RandHound/RandHerd [281]; here, we adapt the approach from Algorand because, unlike RandHound/RandHerd, it can scale to millions of participants.

Briefly, the protocol works as follows. Each round i has a "block" B_i of random bits. The blocks are usually uniformly random from $\mathcal{A}$'s perspective, and $\mathcal{A}$ can only manipulate them within strict limits. B_i determines the committee, as well as a "leader" L_i, as follows. First, each device signs three messages $(B_i, i, 0)$, $(B_i, i, 1)$, and $(B_i, i, 2)$. (The third element in these triples is just to ensure that the hashes of the messages are independent.) The committee then consists of the devices whose signatures on $(B_i, i, 0)$ have the lowest hash, the "leader" is the device whose signature of $(B_i, i, 1)$ has the lowest hash, and the next random number B_{i+1} equals the hash of the leader's signature of $(B_i, i, 2)$.

A detailed description of the protocol, which we call GET_NEW_COMMITTEE, is in the figure below. As part of the protocol, $\mathcal{A}$ maintains a Merkle tree [211] of an array of registered devices. This allows it to publish a constant-sized tree root that is bound to the state of the array at a given point in time, and subsequently to provide logarithmic-sized proofs that devices are in the committed array [24]. We assume that B_0 is a random number that is provided by a trusted source *after* the set of initial devices, $R_{-\infty}$, is already committed to by placing the tree root on the bulletin board B.

Every time a device sends $\mathcal{A}$ a message, $\mathcal{A}$ must send a signed acknowledgment of having received the specific message. If $\mathcal{A}$ fails to do so, the device reports through the reporting channel, X, that it has not yet received a message that is due from $\mathcal{A}$, giving $\mathcal{A}$ an opportunity to respond publicly. If she does not, the device reports that $\mathcal{A}$ has deviated from the protocol. This prevents $\mathcal{A}$ from ignoring devices, in particular devices that should be

leaders or committee members.

GET_NEW_COMMITTEE

1. Each new device who wishes to join registers its public key with $\mathcal{A}$. A device is only eligible to be a leader or committee member if it has been registered for at least κ rounds or was an initial device. $\mathcal{A}$ adds each new key to the set R_i. $\mathcal{A}$ creates a Merkle tree of R_i and posts the root to the bulletin board. This will allow $\mathcal{A}$ to generate proofs $\mu_{i',j}$, that a device j is eligible for election in round i', by showing that $j \in R_t$ for some $t \leq i' - \kappa$.
2. Each device $j \in R_t$ for some $t \leq i - \kappa$ computes $\eta_{i,j,0} = \mathsf{sign}_{sk_j}(B_i, i, 0)$ and sends it to $\mathcal{A}$.
3. $\mathcal{A}$ computes $h_{i,j,0} = \text{Hash}(\eta_{i,j,0})$ for each j. The devices with the C lowest $h_{i,j,0}$ form the committee. $\mathcal{A}$ posts the committee, along with their $\eta_{i,j,0}$ and $\mu_{i,j}$, to the bulletin board.
4. Each device j that is in set R_t for $t \leq i - \kappa$ computes $\eta_{i,j,1} = \mathsf{sign}_{sk_j}(B_i, i, 1)$ and sends it to $\mathcal{A}$.
5. $\mathcal{A}$ computes $h_{i,j,1} = \text{Hash}(\eta_{i,j,1})$ for each j. The device with the minimum $h_{i,j,1}$ is the leader L_i. $\mathcal{A}$ posts $(L_i, \eta_{i,\ L_i,\ 1}, \mu_{i,\ L_i})$ to the bulletin board.
6. The leader L_i sends $\eta_{i,\ L_i,\ 2} = \mathsf{sign}_{sk_{L_i}}(B_i, i, 2)$ to $\mathcal{A}$, who posts it on the bulletin board. Then $B_{i+1} = \text{Hash}(\eta_{i,\ L_i,\ 2})$.
7. If the leader does not respond in time, then $B_{i+1} = \text{Hash}(B_i, i)$.
8. Each device j checks that:
   - If j is not on the committee, then $h_{i,j,0} > h_{i,k,0}$ for each committee member, k.
   - $\mu_{i,k}$ is correct for each committee member k.
   - For each committee member, k, $\eta_{i,k,0}$ is a correct signature for k.
   - If $j \neq L_i$ then $h_{i,j,1} > h_{i,L_i,1}$.
   - $\mu_{i,\ L_i}$ is correct.
   - $\eta_{i,\ L_i,1}$ and $\eta_{i,\ L_i,\ 2}$ are correct signatures for L_i.

If any of these fail, the device sends the evidence of the failure to the reporting channel, X, and aborts the protocol.

A full proof of how this ensures the unlikeliness of a malicious committee is provided in Appendix C.3.

4.3.3 Setup phase: Key generation

Once a new committee has formed, the committee members must generate a new keypair (PK, SK) for the homomorphic cryptosystem that will be used to encrypt and aggregate the users' private data records for this round. As explained in 4.3.1, the secret key will be

stored using a secret sharing scheme. It will remain safe, as long as there are fewer than $\frac{2}{5}C$ malicious committee members. Additionally, the scheme must be able to detect if malicious committee members attempt to introduce an error into the secret during reconstruction, provided fewer than $\frac{2}{5}C$ of them collaborate to attempt this. Lastly the scheme should allow for up to $\frac{1}{5}C$ committee members to go offline. We achieve this with Shamir Sharing [270], based on the Reed-Solomon code [253], with parameter $t = \lfloor \frac{2}{5}C \rfloor$.

The following protocol, KEY_GEN, is performed within the MPC to securely generate a key-pair:

KEY_GEN

1. Choose $(PK, SK) \xleftarrow{r} KeyGen()$
2. Publicly reveal PK to all participants.
3. Distribute SK using a secret-sharing scheme that detects errors when there are fewer than $\frac{2}{5}C$ errors and is secure against up to $\frac{1}{5}C$ erasures.

4.3.4 Collect phase: Querying

Honeycrisp may run for a long time, and during that time the needs of the aggregator could change; thus, it could be problematic to hard-code a specific query, or set of queries, in the design. Instead, Honeycrisp can optionally support a simple query language that can be used to specify arbitrary queries over the data that is available at each device. For instance, the aggregator could ask for a count-mean sketch of emoticons today, and a count of the devices that have shut down because of low battery [19] tomorrow. The question of what to include in the "database" that is available for querying is up to the operator; users could also be allowed to enable or disable certain items based on their own preferences.

Since Honeycrisp relies on an additively homomorphic cryptosystem for aggregating the collected records, not all queries can be supported. However, we *can* support counts and sums, as long as we maintain queries that are 1-sensitive for differential privacy purposes. For instance, Honeycrisp can easily compute the number of devices that have a given property, make histograms or count-mean/count-min sketches, and can sum or average values

from a group of devices. These types of queries boil down to two steps: the first, which we call the *map step*, maps each record in the data set to a vector of numeric values (or even a single value), and the second, which we call the *sum step*, then adds up all the vectors to produce the final output.

To verify that a proposed query has a finite "privacy cost" that is within the remaining privacy budget, the committee must be able to determine the sensitivity of the query – that is, the amount by which the answer can change if a single person's data is added or removed. We can enable this by writing the queries in a language such as Fuzz [144], which comes with a static analysis that bounds the sensitivity.

Once the committee has verified that the remaining privacy budget is sufficient for the proposed query, the honest committee members sign a query authorization certificate that includes the public key generated in Section 4.3.3, the query specification, the remaining privacy budget, and the current round, and they upload it to the aggregator, which distributes it to the other devices. The other devices verify that the certificate has been signed by at least $\frac{2}{5}$ of the current committee (whose membership they know from Section 4.3.2); if so, they accept the included public key and query.

4.3.5 Collect phase: Aggregation

Once a device has received the certificate and the query from the aggregator, and once it has verified the certificate, it locally performs the query's map step – using the data on that particular device – and obtains a vector of numeric values.

At first, it may seem that the device can simply encrypt the values using the homomorphic cryptosystem and send them to the aggregator. However, this is not enough to guarantee privacy. While $\mathcal{A}$ will not be able to learn any information from the ciphertext itself, $\mathcal{A}$ may, during a Byzantine period, send to the committee an incorrect aggregation, such that the query result exposes sensitive user information. For instance, if $\mathcal{A}$ used the additively homomorphic property to multiply a single device's input x_i by a sufficiently large constant, then the result of the query would allow conclusions about x_i, since the Laplace noise will be

"too small" to hide such a large contribution. Alternatively, $\mathcal{A}$ could create Sybil identities and choose the inputs of these identities to be x_i as well – which $\mathcal{A}$ can do because the ciphertext c_i is uploaded to it. To prevent attacks such as these, we would like $\mathcal{A}$ to prove that 1) each honest device's input affects only its own ciphertext, and that 2) the summation was correctly computed.

To prevent adaptive choices of ciphertexts, Honeycrisp requires that all inputs to the summation be committed to before any are revealed. It also requires that the summation process is checked. Since the number of devices is too large for the entire summation to be checked by any device, $\mathcal{A}$ generates, and commits to, an object we call a *summation tree* that contains the inputs and partial sums. This is done using the AGGREGATE protocol below.

AGGREGATE

1. Each device holds a key-pair (σ_i, π_i) for a signature scheme and a private input x_i.
2. Each device computes $c_i = Enc_{PK}(x_i)$
3. Each device generates a commitment to (c_i, π_i), namely $t_i = \text{Hash}(r_i \,||\, c_i \,||\, \pi_i)$, where $r_i \xleftarrow{r} \{0,1\}^{128}$. The device sends (π_i, t_i) to $\mathcal{A}$.
4. $\mathcal{A}$ sorts pairs (π_i, t_i) by π_i to form an array of tuples *Commit*. $\mathcal{A}$ generates a Merkle tree M_C from array *Commit* and publishes the root to B.
5. Each device generates a zero-knowledge proof, z_i, that the plaintext x_i that corresponds to the ciphertext c_i is in the required range.
6. Each device sends (π_i, c_i, r_i, z_i) to $\mathcal{A}$.
7. $\mathcal{A}$ checks the message. If either the proof, z_i, or the commitment, $t_i = \text{Hash}(r_i||c_i||\pi_i)$, is wrong, they ignore the message.
8. $\mathcal{A}$ generates a summation tree, S. The leaves are set to be $S(0,i) = (\pi_i, c_i, r_i)$ if $\mathcal{A}$ received a correct message from a device and $(\pi_i, \perp)$ otherwise. Each non-leaf vertex has two children and a ciphertext that is the sum of its children's ciphertexts.
9. $\mathcal{A}$ serializes all vertices of S into an array and then publishes a Merkle tree M_S of this array, as well as the root of the summation tree S, (the sum of all ciphertexts). To each device sent a correct leaf, $\mathcal{A}$ sends a proof that it is in M_S.

Each device checks a small random portion of this tree, using the CHECK_AGGREGATION protocol. Devices can check that an item is in the set by asking $\mathcal{A}$ to sending a membership

proof for the item which consists of $\lceil \log N \rceil + 1$ hashes from the Merkle tree [211]. If no device reports a problem, this means that, with high probability ($\geq 99\%$, based on a security parameter s), the entire summation is correct.

Notice that the protocol also requires the devices to prove, in zero knowledge, that their inputs are in the correct range – e.g., using a zk-SNARK [40]. This step is not necessary for privacy, but it is necessary for integrity: without it, a single malicious device could render the entire query result useless by encrypting and submitting a very large random value.

CHECK_AGGREGATION

Each device:

1. Verifies that $N \leq N_{max}$, and that the value $Commit_i$ it sent to $\mathcal{A}$ appears in M_C.
2. Chooses a random $v_{init} \in [0, N-1]$. Then for $i \in [v_{init}, v_{init} + s] \bmod N$, verifies that:
   - $Commit_i$ appears in M_C
   - $S(0, i) = (\pi_i, \perp)$ or (π_i, c_i, r_i).
   - If $c_i, r_i \neq \perp$, checks $t_i = H(r_i || c_i || \pi_i)$.
   - $S(0, i)$ appears in M_S.

   Then for $i \in [v_{init}, v_{init} + s) \bmod N$ checks that $\pi_i < \pi_{i+1}$ (except if $i = N - 1$).
3. Chooses s distinct non-leaf vertices of S. To reduce redundancy, this should include the (roughly $s/2$) vertices whose children the device has already obtained from the previous step. The remaining vertices should be chosen randomly from vertices that do not have leaves as children. For each vertex, the device verifies:
   - That the vertex's ciphertext is indeed the sum of its childrens' ciphertexts.
   - That the vertex and its children are in M_S.

If any check fails, the device publicly publishes to X the proof that $\mathcal{A}$ behaved maliciously (signed claims from $\mathcal{A}$ that are inconsistent).

4.3.6 Test phase: Key recovery

At the end of the collect phase, the aggregator has obtained the encrypted true result of the query. Next, the result must be decrypted and compared to the analyst's "guess." Since no individual party can be trusted with the full private key, the committee must run another MPC to do the decryption.

The aggregator submits the vector of ciphertexts and the analyst's guess(es) to the committee. The committee members then launch a multi-party computation to which they

each input 1) their share of the private key, 2) the ciphertext from the aggregator, 3) the analyst's guess, and 4) a threshold difference, below which variation of the guess from the actual result will not be revealed. Inside this computation, the private key is reconstructed from the shares, and is then used to decrypt the ciphertext(s). If too many committee members have gone offline since the beginning of the round or now refuse to participate, the MPC run fails and the aggregator does not receive a response to her query for that round.

4.3.7 Test phase: Noising

Once the encrypted sums have been decrypted inside the MPC, some noise must be added to the plaintext values *before* they are compared to the analyst's guess. (This is part of the SVT.) The noise must be drawn from a distribution which gives correct differential privacy guarantees, and there must not be a way for a malicious committee member to bias the noise in any way. Often, random noise drawn from a Laplace distribution with parameter $(\Delta f/\epsilon)$ is used to support Δf-sensitive queries, as this guarantees $(\epsilon, 0)$ differential privacy. In our case, we simply support 1-sensitive queries to make use of the sparse vector mechanism, so we fix $\Delta f = 1$. The amount of noise will be a fixed amount set by the MPC in Honeycrisp based on the pre-determined privacy budget, such that no party (either the aggregator or the committee members) has any ability to affect the privacy guarantees. One additional concern is the existence of floating-point vulnerabilities that may arise from irregularities in existing implementations of the Laplacian mechanism that create porous (and thus attackable) distributions. Thus, the noise generation must be carefully implemented (for instance, with a snapping mechanism as described in [215]) to ensure differential privacy and to prevent such attacks.

4.3.8 Test phase: Thresholding

Finally, the noised results are compared to the analyst's guess. Once again, this must be done within the multi-party computation, to prevent individual devices from "leaking" the result to the aggregator. Somewhat counter-intuitively, such a leak would be problematic

even after noise has been added: the privacy budget is not substantially charged if the analyst's guess was correct, so, if the data is stationary, the analyst could run very many queries "for free", average out the noise, and then use the precise result to infer the individual inputs. This requirement means that Honeycrisp cannot use a generic threshold cryptosystem [91] but instead must use more powerful MPC-based approach.

If the difference between the guess and the noised result is larger than the threshold, the computation outputs the noised result, and otherwise a default value to indicate that the guess was approximately correct. In the former case, the committee members deduct the "cost" of the query from the privacy budget and report the noised result back to the aggregator; in the (common) latter case, they simply report the outcome and decrement the large number of "prepaid" negative answers (see Section 4.1.2) but leave the budget itself unchanged.

4.4. Security analysis

A full formal definition of the security requirements, as well as proof that Honeycrisp meets this requirements is provided in Appendix C. Informally, these properties are:

1. **Privacy.** The system remains ϵ-differentially private for a given ϵ, or else everyone learns, with high probability, that the Aggregator cheated.
2. **Correctness.** When the Aggregator receives a response to a query, that response is correct – that is, the exact answer plus the noise required for ϵ-differential privacy.
3. **Liveness.** As long as there is sufficient privacy budget left, the Aggregator will continue to be able to query the system and receive responses with high probability.
4. **Indemnification.** If the Aggregator follows the protocol, devices cannot fabricate evidence that would prove that the Aggregator had deviated from the protocol.

4.5. Implementation

In this section, we give a quick overview of the Honeycrisp prototype we used for our experimental evaluation. The code is available under an open-source license at `https://github.com/danxinnoble/honeycrisp`.

Shamir secret sharing: We use the error-correction properties of Shamir sharing [270] to tolerate the possibility that after key generation, some committee members' devices go offline before the second MPC protocol. Thus the output of the key-generation protocol MPC is a Shamir sharing of the secret key among the k committee members such that any subset of size $t + 1$ can reconstruct the secret, and such that no t nodes can learn anything (in an information-theoretic sense) about the secret. Shamir sharing also has the property that if there are at least $t+1$ honest nodes, the honest nodes can detect any errors introduced by dishonest nodes.

MPC: Our implementation focused on the major computational bottlenecks for our systems – the two MPC protocols. We implemented the MPC protocols using the SCALE-MAMBA framework [176]. SCALE-MAMBA is a compiler and virtual machine for running generic MPC computations. It is the successor to the SPDZ framework [82], and is based on many of the same protocols. SCALE-MAMBA is very well suited for our application: it is truly multiparty (able to compute an MPC between any number of parties), it is secure against malicious adversaries who deviate from the protocol, and it allows developers to express functions using familiar high-level programming syntax rather than boolean or arithmetic circuits.

Because SCALE-MAMBA provides Shamir-sharing as one of its built-in MPC sharing schemes, we were able to use this native scheme to store the secret key between the key generation and decryption rounds. We modified the open-source SCALE-MAMBA source code to reconstruct the secret key automatically using existing shares, even if some of the nodes went offline between the key generation and the decryption.

SCALE-MAMBA operations are performed in a finite field modulo a prime p. This complements our Ring-LWE encryption scheme particularly well, since we could use p as the integer modulus for the LWE scheme. This meant that native SCALE-MAMBA operations were automatically modulo p, so we did not need to implement the modular arithmetic within the MPC. Furthermore, SCALE-MAMBA allows this prime modulus to be config-

ured. In Ring-LWE, the additive homomorphism of plaintexts is modulo some integer q, where $|p \mod q| \ll q$, ideally $p = 1 \mod q$. Being able to specify p allowed us to have a sufficiently large plaintext modulus to hold the aggregation.

Ring-LWE: Honeycrisp requires an additively homomorphic cryptosystem to aggregate user inputs, and we instantiate our scheme using the simple "two-element" Ring-LWE-based encryption scheme of [200]. We chose this encryption scheme because its key generation and decryption operations are very simple algebraically – each involves a small constant number of additions and one multiplication in the ring $\mathbb{Z}_p[x]/(x^n + 1)$ where p is prime and n is a power of 2.

The encryption scheme works over a polynomial ring $R_p \stackrel{\text{def}}{=} \mathbb{Z}_p[x]/(x^n+1)$. Then the secret key is a random polynomial $s(x) \in R_p$, and the public key is a pair generated by sampling a random $a \in R_p$ and setting the public key to be $(a, b) \in R_p^2$, where $b \stackrel{\text{def}}{=} a \cdot s + e \in R_p$, for some "error" $e \in R_p$ chosen from an appropriate error distribution. The plaintext space is $\mathbb{Z}_q^l$, where $q, l \in \mathbb{Z}$, $l \leq n$, $q \ll p$ and $|p \mod q| \ll q$. To encrypt a vector $z \in \mathbb{Z}_q^l$, the encryptor generates a random $r \in R_p$, and computes the ciphertext $(u, v) \stackrel{\text{def}}{=} (a \cdot r + e_1, b \cdot r + \lfloor p/q \rceil \cdot z) \in R_p^2$. Decryption is then simply $z = \mathit{round}(v - u \cdot s, \lfloor p/q \rceil)/\lfloor p/q \rceil$ where $\mathit{round}(x, y)$ rounds each coefficient of x to the nearest multiple of y. (This assumes the errors e, e_1, e_2 are sufficiently small relative to p/q).

Our design and implementation for Ring-LWE key generation and decryption inside of an MPC was developed independently from the concurrent work of [173], except that we use their observation that if the plaintext length, l, is less than n, then only l coefficients of v ever need to be stored.

Security parameters: We use the LWE-estimator tool [198] of Albrecht et al. [15] to obtain concrete parameters that provide sufficiently high security based on the best current LWE attack algorithms. Using this tool, we find that dimensionality $n = 4096$, a 128-bit prime p, and a Gaussian error distribution with $\sigma = \frac{\sqrt{2}}{2}$ (which we approximate as the

centered binomial distribution with $N = 2$ trials) in each dimension, provides over 128 bits of security.

We note that there is a space-time tradeoff: on the one hand, Ring-LWE's easy decryption and key generation simplify the committee's MPCs, and the large dimension allows many metrics to be aggregated in parallel – while our implementation only uses one counter, our choices can yield up to 4,096 counters, each with a capacity of about 50 bits! But on the other hand, the ciphertexts are fairly large, which increases the bandwidth cost of the aggregator (Section 4.6.5). With a different homomorphic encryption scheme, such as Paillier [234] or elliptic-curve-based El Gamal (ECEG), the MPCs would take longer, but the ciphertexts would be smaller.

The verification portion of our scheme requires a collision resistant hash function (for the Merkle Trees) and a signature scheme for each user. Following standard practice, we use a SHA-256 hash function and RSA-2048 signatures.

4.6. Evaluation

Our goal for the experimental evaluation is to answer the following three questions: 1) Can Honeycrisp support periodic queries while giving reasonable privacy guarantees?, 2) How expensive is Honeycrisp in terms of computation, bandwidth, and storage?, and 3) How well does Honeycrisp scale?

4.6.1 Experimental setup

Honeycrisp is designed to operate in a very large deployment with potentially billions of laptops and phones, as well as a large data center. Since we did not have access to a large enough testbed, we benchmarked several of the components individually. For user-side computations, this is safe, since users communicate only with the aggregator and not with each other, and for the aggregator's computations we can easily extrapolate the cost because the operations are simple and can mostly be done in parallel. The only component of Honeycrisp that requires more attention is the committee; here, we cannot simply extrapolate, but fortunately the committees are small enough for us to run the

corresponding computations completely.

Our aggregator experiments were run on eight PowerEdge R430 servers with 64 GB of RAM, two Xeon E5-2620 CPUs, and 10 Gbps Ethernet. The operating system was Fedora Core 26 with a Linux 4.3.15 kernel. This equipment seems reasonably close to what a real-world aggregator would have in its data center. To simulate users operating in a global setting, we used multiple `t2.large` Amazon EC2 servers with 8 GB of RAM, located in all available geographic regions, to obtain realistic latencies and communication costs.

We compare three different systems: 1) a RAPPOR-style solution (LDP) that achieves differential privacy in the local setting by making use of the randomized response mechanism; 2) a hypothetical solution (GDP) that uploads the unencrypted data to the aggregator, which then releases the result using global differential privacy, but not the SVT, somewhat analogous to PINQ [208]; and 3) our proposed solution, Honeycrisp. Notice that the second solution cannot protect user privacy against the aggregator, so it is not necessarily a realistic comparison point for Honeycrisp; nonetheless we demonstrate an improvement over this generic setting.

4.6.2 Utility

Our first goal is to determine whether Honeycrisp really can support queries for longer than existing systems. To this end, we simulate a comparison over a 10-year span between LDP, GDP, and Honeycrisp. We consider a simple vector of sensitivity-1 counting queries, which is at the heart of the count-mean sketches Apple is using [22], and we assume that the query needs to be asked once per day. Our model query is performed on a corpus of Twitter data spanning 5 years, and the count-mean sketch is over word usage frequency for newly-appearing words in the English language. We assume $N = 1.3 \cdot 10^9$ users, which was the size of Apple's deployment in February 2018 [18], and we choose the parameters in such a way that the total, noised count is within 1% of the true count with probability $p = 0.95$, assuming a query with a constant fraction that .001% of users respond to (although this error is a constant factor that affects all systems identically). For Honeycrisp, we set a

threshold of 5%, and we (conservatively) assume that, on average, the true count changes by that amount about once every three months. This seems realistic: research on changing use of out-of-vocabulary language, as well as our own queries, show that word frequency changes as little as $1 - 1.5\%$ over an entire year [108].

Figure 2 shows a simulation of the privacy budget consumption of all three systems over time. The LDP-based system has the highest consumption by far; it goes through a budget of $\varepsilon = 1$ (a common choice [156], indicated by the horizontal line) approximately every 91 days. This is because, in the local setting, each user's data must be noised individually, so the sum contains much more noise than with global differential privacy, where the sum is computed precisely and then noised only once. As discussed in [106], with n this results in an incurred error cost of $O(\sqrt{n})$, as opposed to $O(1)$ in the global setting. This is consistent with Apple's decision to renew the privacy budget very frequently, and if more users were to respond to every query, this cost would become even higher! The consumption of the (insecure and hypothetical) GDP-based system is lower, but a budget of $\varepsilon = 1$ would last less than half as long as Honeycrisp over this time span, when we consider both systems operating over data vectors of size 10. This is because Honeycrisp has a second advantage: with the sparse-vector technique, the budget decreases logarithmically with the total number of queries (as opposed to linearly) and needs to be charged substantially only in the case when the answer changes. Because of this, Honeycrisp can run for 10 years without exhausting its privacy budget of $\varepsilon = 1$. For additional details, please see Appendix A.

Note that at first, the bound on the privacy budget for Honeycrisp is *higher* than that of both the LDP and GDP. This is because of the way the SVT works: it charges a relatively large privacy cost at the beginning, based on the expected number of times the data will change, and then charges only logarithmically for queries where the analyst's estimate turns out to be approximately correct. (The cost for such queries is not exactly zero because the threshold comparison is performed on the already-noised answer, so there will be occasional charges even when the estimate is correct.) In contrast, the other systems' privacy budgets degrade

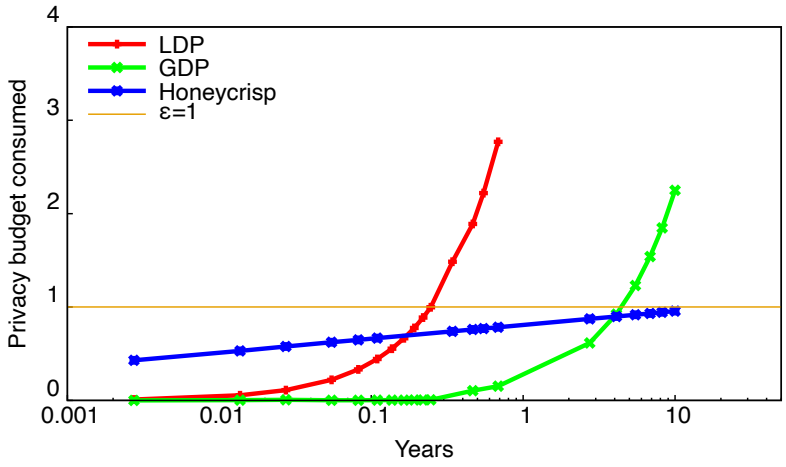


Figure 2: Budget consumption over time.

linearly, so, in the long run, Honeycrisp uses its privacy budget much more efficiently.

Even at this much lower rate of consumption for Honeycrisp, any finite privacy budget will eventually run out. However, "recharging" the privacy budget is not unreasonable per se, since many secrets would become far less valuable to an adversary if it took years to learn them. The key question is how frequently the budget needs to be recharged, and here Honeycrisp outperforms basic randomized response in the local setting by a factor of over 40.

4.6.3 Cost: Normal participants

Next, we examine the cost that a "normal" participant would pay to be part of Honeycrisp. We measured these costs by running all the participant-level steps in a single round of the protocol; we report the storage, bandwidth, and computation time for five system sizes: $N = 1.3 \cdot 10^9$ (the estimated size of Apple's deployment), as well as, for comparison, values ranging from $N = 1.3 \cdot 10^8$ to $N = 1.3 \cdot 10^{10}$.

Bandwidth: Figure 3(a) shows the amount of bandwidth that is consumed in a single round. The amount grows slightly with the system size because the MHT becomes taller and thus its inclusion proofs become longer (with $O(\log N)$). However, at less than 1.2 MB,

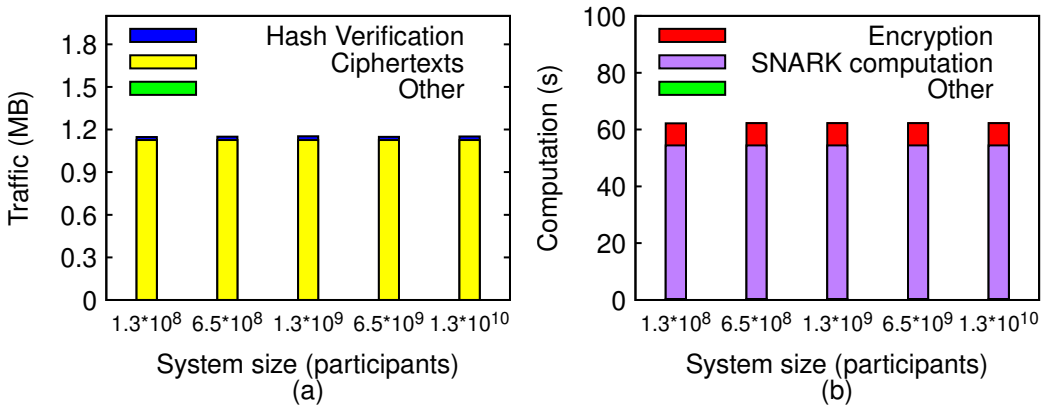


Figure 3: Bandwidth (a) and computation (b) required of each participant in each round.

the overall amount is reasonable even for the largest system size we tried. The commitments and range proofs (in particular, zk-SNARKs) each require less than 1 kB [40], which is too little to be visible in the figure.

Computation: Figure 3(b) shows the amount of computation that a participant needs to perform in each round, in terms of milliseconds of computation time on an E5-2620 core. Checking the signatures on the certificate and the MHT inclusion proof consumes only a small amount of time; the overall amount is likely dominated by the prover's computation. The implementation of [40] has proof times of approximately 0.2 ms per arithmetic gate. Considering the size of the arithmetic circuit implementing our RLWE encryption scheme, this would result in a proving time of approximately 54 seconds. Although this cost is high, each device would need to perform this step only once per query. At one query per day, this should be manageable, especially if (as in our motivating scenarios) quick turnaround times are not required and the computation can be done slowly in the background.

Storage: Participant machines do not need to permanently store any information, since they can always download the entire history of blocks and Merkle-tree roots from the bulletin board. However, it makes sense to store at least the most recent randomness block, most recent certificate, and at most 3 ciphertexts at a time for summation verification, which together would be less than 200 kB.

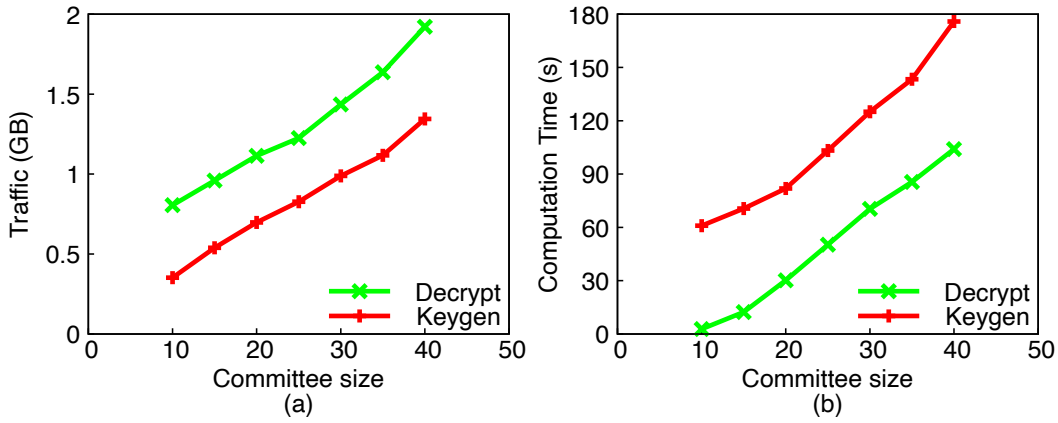


Figure 4: Bandwidth (a) and time (b) required for each of the two MPC steps using SCALE-MAMBA, for a RLWE cryptosystem in a lattice of dimension $n = 4096$.

4.6.4 Cost: Committee

We now quantify the cost of a participant that has been chosen as a committee member for the current round. Such a participant must perform two additional steps: 1) the MPC to generate the keypair, as discussed in Section 4.3.3, and 2) the MPC to decrypt, noise, and threshold the aggregate, as discussed in Section 4.3.6. (There are other small costs, such as signing the certificate, but we ignore them here because the MPC costs clearly dominate.) These costs are independent of the number N of participants, but they do very much depend on the committee size, which is why we vary this parameter from 10 to 40 users.

In Figure 4(a), we show the total number of bytes that are sent by a committee member in each of the two MPCs; Figure 4(b) shows the total completion time for each MPC. We see that both time and traffic scale linearly with the size of the committee. With $C = 40$ committee members, each committee member uses less than 5 minutes and about 3.3 GB for both protocols combined. The MPC execution consists of an online phase and a secure pre-processing phase that generates randomness. The latter is responsible for much of the cost, making it difficult (but not impossible) to run this process on mobile devices.

If the cost is too high for the mobile devices, there are at least two possible solutions. One is

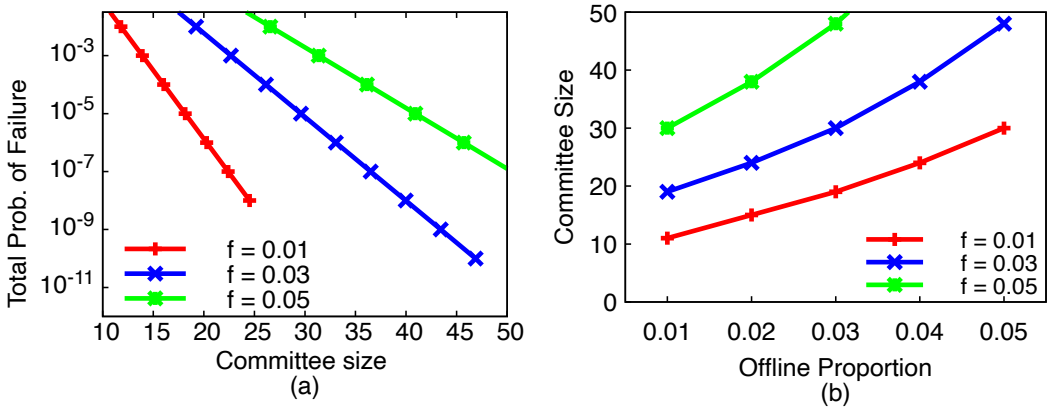


Figure 5: a) Probability of privacy failure for various committee sizes; b) minimum committee size needed for liveness.

to avoid mobile devices entirely and to ask only more powerful devices (laptops or desktops) to serve on the committee. If the adversary cannot target specific device types, this merely results in a smaller pool of potential committee members. If the adversary *can* target the candidate devices specifically, this approach would require us to scale down the fraction f of malicious devices; for instance, if we assume $f = 3\%$ but two thirds of the devices are mobile, we would need to choose the other parameters based on $f = 1\%$ instead. The other way is to leverage a party with limited trust, if one happens to be available. We do not discuss this option here due to lack of space, but in Appendix B, we show that it can reduce the cost to almost zero. Our experiments with the SPDZ multiparty compiler show that, in this case, the online phase alone requires just 10 MB for both protocols combined.

Next, we justify our choice of committee sizes. Figure 5(a) shows the probability of a privacy failure during a 10-year period, given various settings for the fraction of malicious nodes f and the committee size C. With $f = 3\%$ and a committee of $C = 40$ members, the chance of *ever* seeing a privacy failure (that is, a committee with too many malicious nodes) during the ten years is about 10^{-8}. Figure 5(b) similarly shows, for various settings of f and the fraction of offline nodes g, the minimum committee size that would be needed to ensure that at least 95% of the queries receive an answer. Again, with $f = 3\%$ and $g = 4\%$, a committee of $C = 40$ members would be sufficient. Notice that, if more nodes

are offline than the choice of g anticipates, the result is simply that a few more queries will go unanswered.

4.6.5 Cost: Aggregator

Finally, we turn our attention to the aggregator. The aggregator clearly has the highest workload, but it also presumably has the most resources. Since we cannot fully replicate the aggregator in our lab, we benchmark the various steps individually and then extrapolate. As before, we focus on an estimated size of $N = 1.3 \cdot 10^9$, as well as, for comparison, values ranging from $N = 1.3 \cdot 10^7$ to $N = 1.3 \cdot 10^{10}$.

Bandwidth: The aggregator would need to receive, from each client, a public key and a ciphertext. (We ignore the single copy of the certificate and the final result that the aggregator receives from the committee because they are insignificant.) The aggregator would need to send, to each client, a MHT inclusion proof, a copy of the committee's certificate, and a selection of ciphertexts for summation tree verification, using $s = 5$, giving 99% verification of correctness (see Section 4.3.5), and thus requiring at most 17 ciphertexts to be sent to each user. The only variable-size items are the inclusion proofs, which require $N \log N$ bytes given N participants, and the number of ciphertexts, which scales linearly; the public keys are 256 bytes each, the ciphertexts 65,552 bytes each, and the certificates 92 bytes each using an RSA certificate.

Figure 6(a) shows the total amount of bandwidth (bytes sent or received) that the aggregator would need in each round. Overall, the bandwidth consumption grows with $O(N \log N)$, with a strong linear component. At $N = 1.3 \cdot 10^9$, the amount sent would be roughly 1450 TB, or 1.12 MB/user; for comparison, this would be less than the amount of traffic generated by having 60% of the users download a typical web page (about 2 MB [60]) from the aggregator. If the traffic is a concern, it could be reduced to about 1.45 TB by using ECEG instead of Ring-LWE, at the expense of somewhat longer MPCs for the committee, as discussed in Section 4.5.

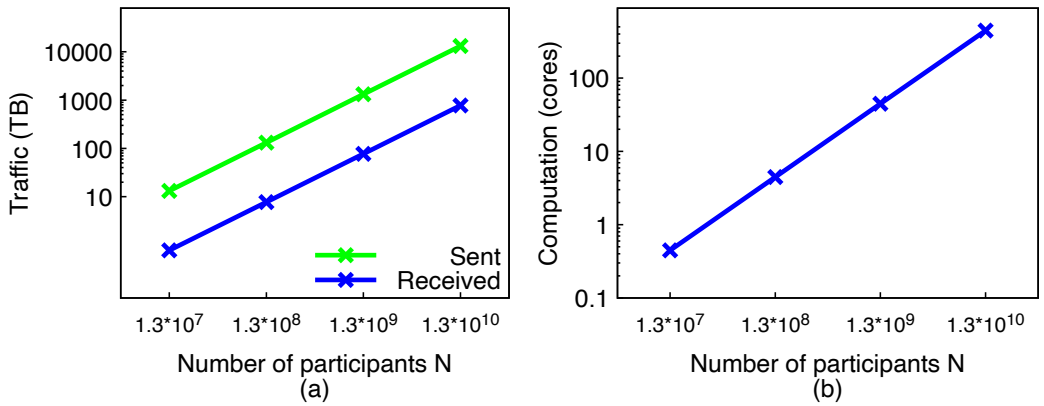


Figure 6: Bandwidth (a) and computation (b) required for the aggregator.

Computation: The aggregator would need to generate the MHT, verify the range proof that each participant uploads, and perform the homomorphic addition. (The inclusion proofs do not require extra work because they can simply be read from the MHT once it is generated.) With our choices for the hash function (SHA-256) and the homomorphic cryptosystem (Ring-LWE), a single hash operation takes 0.005 ms and a single homomorphic addition takes 1.7 ms. For the range proofs, we estimate a verification cost of 5 ms, based on [40].

Figure 6(b) shows the total computation cost in terms of computation time on a single E5-2620 core. If we (somewhat arbitrarily) require the computation phase of each round to last no more than an hour, the aggregator would need 45 cores for $N = 1.3 \cdot 10^9$, which seems achievable.

Storage: The aggregator would need to store the public keys and ciphertexts of all the participants and the MHT. (The range proofs can be discarded once verified.) With 2048-bit keys, SHA-256 hashes, and LWE encryption, a public key, a single hash, and a ciphertext consume 256 bytes, 32 bytes, and 65,552 bytes respectively, so the overall storage requirement is 65.84 kB per user, or roughly 86 TB for $N = 1.3 \cdot 10^9$. Again, this seems clearly within the power of a typical aggregator.

4.7. Conclusion

Honeycrisp fills a gap in the space of secure aggregation systems: it can stretch a given privacy budget much longer – possibly over as much as ten years – as long as the underlying data does not change too often, and it does so in a highly scalable way, without introducing a trusted party. Thus, Honeycrisp could help to address the criticism of existing deployments, e.g., the one operated by Apple, by addressing the unique threat model that these data aggregators face. Honeycrisp does require a nontrivial amount of computation from the (small) group of user devices that is serving on the committee, but the recent improvements in MPC implementations (e.g., [56, 176, 292]) make it seem likely that this cost can be further reduced in the coming years.

CHAPTER 5 : Orchard

The Honeycrisp system described in Chapter 4 can provide global differential privacy at scale, with a single, untrusted aggregator. Instead of fully homomorphic encryption, Honeycrisp uses additively homomorphic encryption, which is much more efficient. However, the price to pay is that Honeycrisp can answer only one specific query, namely count-mean sketches [22] with additional use of the sparse-vector operator. This query does have important applications (for instance, it is used in Apple's iOS), but it is by no means the *only* query one might wish to ask: the literature is full of other interesting queries that can be performed with global differential privacy (e.g., [48, 113, 142, 143, 206, 236, 247, 299]). Right now, we are not aware of any systems that can answer even one of these queries at scale, using only a single, untrusted aggregator.

In this chapter, we show how to substantially expand the variety of queries that can be answered efficiently in this highly distributed setting. Our key insight is that many differentially private queries have a lot more in common than at first meets the eye: while most of them transform, group, or otherwise process the input data in some complicated way, the heart of the algorithm is (almost) always a sequence of sums, each computed over some values that are derived from the users' input data. This happens to be exactly the kind of computation that Honeycrisp's collect-and-test (CaT) primitive can perform efficiently, using additively homomorphic encryption. Thus, CaT turns out to be far more general than it may seem: it can perform the distributed parts of many queries, leaving only a few smaller computations that can safely be done by the aggregator, or locally on each user device.

The key challenge is that, for many queries, the connection to sums over per-user data is far from obvious. Many differentially private queries were designed for a centralized setting where the aggregator has an unencrypted data set and can perform arbitrary computations on it. Such queries often need to be transformed substantially, and existing operators need

to be broken down into their constituents, in order to expose the internal sums. Moreover, a naïve transformation can result in a very large number of sums—often far more than are strictly necessary. Thus, optimizations are needed to maintain efficiency.

We present a system called Orchard that can automatically perform these steps for a large variety of queries. Orchard accepts centralized queries written in an existing query language, transforms them into distributed queries that can be answered at scale, and then executes these queries using a generalization of the CaT mechanism from Honeycrisp. Among 17 queries we collected from the literature, Orchard was able to execute 14; the others are not a good fit for our highly distributed setting and would require a different approach.

Our experimental evaluation of Orchard shows that most queries can be answered efficiently: with 1.3 billion users (roughly the size of Apple's macOS/iOS deployment [18]), most user devices would need only a few megabytes of traffic and a few minutes of computation time, while the aggregator would need about 900 cores to get the answer within one hour. For queries that make use of the sparse-vector operator, this is competitive with Honeycrisp; for the other queries we consider, we are not aware of any other approach that is practical in this setting. In summary, our contributions are:

- the observation that many differentially private queries can be transformed into a sequence of noised sums (Section 5.1);
- a simple language for writing queries (Section 5.2);
- a transformation of queries in this language to protocols that can answer them in a distributed setting, using only a single, untrusted aggregator (Section 5.3);
- the design of Orchard, a platform that can efficiently execute the transformed queries (Section 5.4);
- a prototype implementation of Orchard (Section 5.5); and
- an experimental evaluation (Section 5.6).

5.1. Overview

Scenario: We consider a scenario (as in previous chapters) with a very large number of users (millions), who each hold some sensitive data, and a central entity, the *aggregator*, that wishes to answer queries about this data. We assume that each user has a device (say, a cell phone or a laptop) that can perform some limited computations, while the aggregator has access to substantial bandwidth and computation power (say, a data center).

Threat model: We make the OB+MC assumption from 4.1.1—that is, we assume that the aggregator is honest-but-curious (HbC) when the system is first deployed and usually remains HbC thereafter, but may occasionally be Byzantine (OB) for limited time periods; for instance, the aggregator could be a large company that is under public scrutiny and would not violate privacy systematically, but may have a rogue employee who might tamper with the system and not be discovered immediately. For the users, we assume that most of them are correct (MC) but that a small percentage—say, 2–3%—can be Byzantine at any given time. This is different from the typical assumption in the BFT literature, where one often assumes that up to a third, or even half, of the nodes can be Byzantine. However, BFT systems are typically a lot smaller than the systems we consider: with 4–7 replicas, compromising a third of the systems means just one or two nodes, whereas, in Apple's deployment with 1.3 billion users, a 3% bound would mean 39 million malicious users, which is much larger than, e.g., a typical botnet.

Assumptions: Our key assumptions are (1) that the approximate number of users is known and (2) that the adversary cannot create and collude with a nontrivial number of Sybils. For instance, the devices could have hardware support for secure identities, such as Apple's T2 chip or Intel's SGX.

Goals: We have four key goals for Orchard:

- **Privacy:** The amount of information that either the aggregator or other users can learn about the private data of an honest user should be bounded, according to the

Query		Support
Decision-tree learning (ID3)	[119]	Yes
k-means	[48]	Yes
Perceptron	[48]	Yes
Principal Component Analysis (PCA)	[48]	Yes
Logistic regression	[3]	Yes
Naïve Bayes	[302]	Yes
Neural Network training (Grad. Descent)	[3]	Yes
Histograms	[299]	Yes
k-Medians	[142]	Yes
Cumulative Density Functions	[206]	Yes
Range queries	[151]	Yes
Bloom filters (RAPPOR)	[113]	Yes
Count Mean Sketch	[22]	Yes
Sparse vector (Honeycrisp)	[263]	Yes
Iterative Database Construction	[143]	No
Teacher Ensembles (PATE)	[236]	No
Vertex programs (DStress)	[235]	No

Table 1: Selection of differentially private queries from the literature, and support by Orchard.

formulation of differential privacy.

- **Correctness:** If all users are honest, the answers to queries should be drawn from a distribution that is centered on the correct answer and has a known shape;
- **Robustness:** Malicious users should not be able to significantly distort the answers; and
- **Efficiency:** Most users should not need to contribute more than a few MB of bandwidth and a few seconds of computation time per query.

5.1.1 Differential privacy

General background on differential privacy is provided in Section 2.1. By now, there is a rich literature on differential privacy proposing many different forms of queries for many different use cases. We have done a careful survey to collect examples that would make sense in our highly distributed setting; Table 1 contains the queries we found, which will also be used in our evaluation (Section 5.6.1).

5.1.2 Alternative approaches

Local differential privacy (LDP): As discussed earlier, another way to avoid trusting the aggregator is to use LDP [113]—that is, for each user to add noise to his or her data individually, *before* uploading it to the aggregator, instead of noising just the final result. However, there are two important challenges. The first is that the noise in the final result now grows with the number of users: for instance, a sum of values from N users now contains N draws from a Laplace distribution $L(\frac{s}{\varepsilon})$, instead of just one! The effective error grows a bit more slowly, with $\Theta(\sqrt{N})$ [106, §12.1], but still, with $N = 10^9$ and $\varepsilon = 0.1$, the median error will be approximately 300,000 with LDP and only 10 with GDP—a difference of several orders of magnitude, which can be severely limiting in practice [46]. The second challenge is that the noise is added by the *users* and not by the aggregator; thus, even a very small number of malicious users can, by using large, correlated values as their "noise" terms, severely distort the final result [73]. We will revisit this problem in Sections 5.4.3 and 5.6.3.

Multiparty computation (MPC): In principle, the data could also be aggregated using MPC [300], a cryptographic technique that enables a group of participants to jointly evaluate a function f such that each participant only learns the final output of f, but not the inputs of each participant. It may seem that all we need to do is set $f := q \circ L(\frac{s}{\varepsilon})$, where q is the query and L is a draw from an appropriate Laplace distribution. The problem, however, is efficiency: generic MPC scales poorly with the number of participants. While there are very efficient solutions for two parties (e.g., [175]) and reasonably efficient ones for a few dozen parties (e.g., [292]), we are not aware of a technique that would be practical with millions or billions of participants.

Fully homomorphic encryption (FHE): With FHE [125], users could encrypt their data with a public key and upload them to the aggregator, who could run the query on the ciphertexts, add noise, and then decrypt only the final result using a private key. As

with MPC, this approach works for arbitrary queries, and it has the advantage that most of the work is done by the aggregator. However, if the aggregator has the private key, it can also decrypt the users' individual uploads—and even if this problem were solved somehow, computation on FHE ciphertexts is still many orders of magnitude slower than computation on plaintexts, so, with a billion participants, this approach does not seem realistic.

5.1.3 Honeycrisp

As discussed in the previous chapter, Honeycrisp can efficiently answer one specific query (namely count-mean sketches) in our setting. As in the hypothetical FHE approach, users encrypt their private data and upload only the ciphertexts to the aggregator; however, there are two critical differences. The first is that Honeycrisp uses *additively* homomorphic encryption, which is orders of magnitude faster than FHE and can be done efficiently at scale. The second is that, to prevent the aggregator from decrypting individual ciphertexts, Honeycrisp delegates key generation and decryption to a small *committee* of 20–40 randomly selected user devices, which uses MPC to perform these (small) tasks. As before, this enables the aggregator to do all of the "heavy lifting" (collecting and aggregating ciphertexts) without ever seeing unencrypted data from individual users; thus, the aggregator does not need to be trusted.

The main drawback of Honeycrisp is that it only supports a single query. Internally, it uses a primitive called Collect-and-Test (CaT), which works roughly as follows (see also Figure 7): each user device computes a vector of numbers, encrypts it with a public key that was generated by the committee, and uploads it to the aggregator, which sums up the ciphertexts using the additive homomorphism. The aggregator then proves to the users that it has computed the sum correctly (which the aggregator, in its Byzantine phases, may not necessarily do); if so, the committee noises and decrypts the final result. This is the primitive that we leverage for Orchard.

Notice that CaT aggregates *vectors*, not just individual numbers. For additively homomorphic encryption, Honeycrisp uses Ring-LWE, which has large ciphertexts that can be

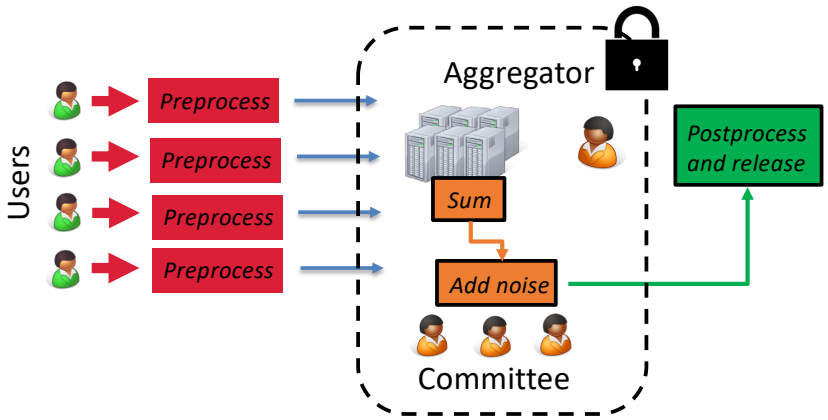


Figure 7: CaT workflow.

subdivided into many smaller fields; these can then be aggregated in parallel. The choices from Honeycrisp yield 4,096 counters with about 50 bits each; thus, a single invocation of CaT can efficiently sum up vectors with thousands of elements. We will leverage this fact for our query optimizations (Section 5.3.5).

5.1.4 Approach and roadmap

Our key insight is that CaT is far more general than it might appear: indeed, the sums it can compute are at the heart of a wide range of differentially private queries. (This is not a coincidence: in fact, a common way to certify differential privacy—e.g., in [31, 88, 123, 144, 254, 301]—is to use a linear type system to track how much a change in a single user's data can affect a given sum or count.) Thus, by rewriting queries to take advantage of CaT, we can considerably expand the range of queries that can be answered at scale. At a high level, Orchard works as follows:

1. The analyst submits her query as a *centralized* program that computes the desired answer based on a (hypothetical) giant database that contains data from all users. Orchard verifies that the query is differentially private (Section 5.2).
2. Orchard transforms this program into a distributed computation that relies on CaT, using several optimizations—such as vectorization—to ensure efficiency (Section 5.3).
3. Orchard executes the distributed program, using protocols from Honeycrisp with some

additional steps, and returns the answer to the analyst (Section 5.4).

5.2. Query language

There are several existing programming languages (e.g., [31, 89, 123, 144, 208, 224, 301, 302]) that can certify differential privacy. Rather than proposing yet another, we adopt an existing language, Fuzz [144]. Fuzz is a functional language, which simplifies our transformations, and its privacy analysis is driven by lightweight type annotations, which is convenient for the analyst. However, the choice is not critical; other languages could be used as well.

5.2.1 Running example: k-means

To conserve space, we introduce the Fuzz language through an example: the widely used k-means clustering algorithm, shown in Figure 8, which will also be our running example for the rest of this chapter. For a more complete description of Fuzz, please see Appendix D.

The k-means algorithm divides a given set of points (the input data) into k clusters and returns a centroid for each cluster. It proceeds in several iterations; for clarity, the figure shows only the iteration step, with k hard-coded to 3. The `step` function is given the current estimates of the centroid positions, `c1`, `c2`, and `c3`, and the set of points `pts`; it first assigns each point to the closest centroid, based on the l_2 distance (`assign`), and then partitions the set of points into three subsets, one for each centroid. Finally, it produces three new centroid positions `c1'`–`c3'` for the next iteration by averaging the coordinates of the points in each subset. This is done by first summing up the coordinates in each partition, and by counting the points; then the `lap` primitive adds Laplace noise to the sums and counts, and then performs the division.

5.2.2 Language features

In most ways, Fuzz is a conventional functional language; just two special features are relevant here. One is that it has a *linear type system*, described in [254], that certifies an upper bound on the sensitivity of all operations on private data; when a noising primitive such as `lap` (for the Laplace distribution) or `em` (for the exponential mechanism) is invoked, the parameter s (Section 5.1.1) is known, and the noise can be drawn from the correct

distribution. The other feature is a *probability monad* that ensures that no private data can "escape" from the program without having passed through `lap` or `em` first. Together, these features ensure that, as long as the top-level program has a type of a certain form, it is guaranteed to be differentially private.

Fuzz encapsulates private data in variables of a special type, `bag`, which represents a set with one element for each individual who contributed data. There are several primitives that operate on bags: `bmap` applies a given function to each element of a bag, `bfilter` removes elements for which a given predicate returns false, and `bpartition` splits a bag into several sub-bags, based on the value a given function returns for each element. All of these primitives take bags as arguments and produce new bags, so the private data remains confined in bags. The final bag primitive is `bsum`, which adds up the elements of a bag.

5.2.3 Alternative languages

Using a language other than Fuzz should not be difficult because the key to Orchard, the basic structure of summing followed by a release mechanism, is present in many other languages for differential privacy. Notice that, in Fuzz, summing via `bsum` is the only way to turn bags into data values that can potentially be released. A similar structure is present, e.g., in PINQ [208], which has three aggregation primitives, of which one (`NoisySum`) is equivalent to `bsum` followed by `lap`; the other two (`NoisyAvg` and `NoisyMed`) are equivalent to `bsum` followed by `em`. Another imperative example, Fuzzi [302], supports the addition of new aggregation primitives through an extension mechanism, but the information we need could be specified as part of the extension. The critical features Orchard needs are 1) a sensitivity analysis and 2) a way to recognize the aggregation primitives in the code.

Another possible approach would be to embed Fuzz as a library into a more traditional data analytics language, such as Python3. This embedded-language approach has already seen success in Deep Learning frameworks, such as TensorFlow [2] and PyTorch [239].

```
assign c1 c2 c3 pt =
  let d1 = sqdist c1 pt
      d2 = sqdist c2 pt
      d3 = sqdist c3 pt
  in if d1<d2 and d1<d3 then 0 else
  if d2<d1 and d2<d3 then 1 else 2

noise totalXY size = do
  let (x, y) = totalXY
  in do x' ← lap 1.0 x
        y' ← lap 1.0 y
        size' ← lap 1.0 size
  return (x'/size', y'/size')

totalCoords pts =
  let ptxs = bmap fst pts
      ptys = bmap snd pts
  in (bsum 1.0 ptxs, bsum 1.0 ptys)

countPoints pts =
  bsum 1.0 (bmap (\pt → 1) pts)

step c1 c2 c3 pts =
  let [p1, p2, p3] =
        bpartition 3 (assign c1 c2 c3) pts
      p1TotalXY = totalCoords p1
      p1Size = countPoints p1
      p2TotalXY = totalCoords p2
      p2Size = countPoints p2
      p3TotalXY = totalCoords p3
      p3Size = countPoints p3
  in do
    c1' ← noise p1TotalXY p1Size
    c2' ← noise p2TotalXY p2Size
    c3' ← noise p3TotalXY p3Size
  return (c1', c2', c3')
```

Figure 8: One step of the k-means algorithm, written in Fuzz. The colors represent the "zones" of computation.

5.3. Query transformation

Next, we describe how Orchard transforms centralized Fuzz queries so that they can be executed in a distributed setting.

5.3.1 Program zones

We begin by observing that, if a Fuzz program is differentially private, it necessarily has a very specific structure and can be broken into three different "zones" (which we color-code in our example in Figure 8):

- **Red zone** computations run directly on the data of an individual user—here, the `assign` function, which finds the closest centroid for each user's data point.
- **Orange zone** computations are performed on user data that has been aggregated

but not yet noised—here, the `lap` operators, which add Laplace noise to the sums.

- **Green zone** computations involve only noised data and constants—here, the final divisions in `noise` and the parts of `iter` that set up the rest of the computation.

The Fuzz type system enforces clear boundaries between these zones: data can only pass from red to orange by aggregation (via `bsum`), and aggregate data can only pass from orange to green by noising (via `lap` or `em`). Moreover, red-zone code always operates on an *individual* element of a `bag`—that is, on data from a a single user. And lastly, none of the operations producing bags offer any way to combine multiple elements of one bag when computing an element of another bag; in other words, every element of every bag that can ever exist is derived (by filtering, partitioning, or mapping) from some single element of some bag that was initially provided as input to the top-level program.

This stratification allows us to map Fuzz programs to Honeycrisp-like computations by mapping the zones to the different parties in Figure 7. Red-zone code is executed directly by user devices; computations in this zone only need the data of one user at a time, so each user device can run it without sending any secrets anywhere. The summation at the red-to-orange boundary can be done as in Honeycrisp, by users encrypting their red-zone outputs and sending them to the aggregator, who adds them up using homomorphic addition and then passes the encrypted sum to the committee. Orange-zone code can be executed by the committee, using MPC, and the members of the committee will be able to decrypt the encrypted sums only after appropriate noise is added. Data that passes from orange to green zones must first pass through a release mechanism (`lap` or `em`) and be thus noised appropriately, so green-zone code can be safely executed "in the clear" by the aggregator itself.

The Orchard compiler uses a special operator to coordinate the mapping, summing, and releasing steps among red, orange and green zones. We call this operator `bmcs` (broadcast, map, clip and sum), and introduce it in the following section.

5.3.2 The `bmcs` *operator*

The operator `bmcs (b,m,c,r)` takes four parameters and behaves as follows:

- first, it *broadcasts* some public state `b` from the aggregator to the user devices;
- on each user device i, it *maps* the local private data d_i to a private vector $v_i := \texttt{m}(\texttt{b}, d_i)$ using the provided map function `m` (which can use the public state in its computation);
- on each user device, it *clips* the elements of v_i such that $|v_{i,k}| \leq \texttt{c}_k$; and finally
- it *sums* all these private vectors from all client devices through homomorphic addition to compute $v := \sum_i v_i$ and returns $\texttt{r}(v)$ using the provided *release function* `r`.

The `bmcs` operator captures the workflow of a single "round" of the distributed protocol; m is the red-zone computation for that round; r is the orange-zone computation. The clipping vector c is needed to guarantee privacy (see Section 5.4.3).

By rewriting a given Fuzz program to use only `bmcs` rather than the individual bag operations `bmap`, `bfilter`, `bsum`, and `bpartition`, we make its "phase-structure" explicit so that we can directly evaluate it on a Honeycrisp-like distributed platform. We next describe how Orchard does this.

5.3.3 Extracting dependencies

When the analyst submits a Fuzz program to Orchard, Orchard begins by reducing complex bag operations (`bpartition` and `bfilter`) into combinations of the two fundamental bag operations—`bmap` and `bsum`. A `bpartition` that splits a bag into k partitions is reduced into a `bmap` that first maps each value in the bag to a partition index, followed by k `bfilter` operations that filters out each of the individual partitions. A `bfilter` operation is reduced into a `bmap` operation that maps each value v in the bag to an optional value v'—when the filter predicate evaluates to true on v, the optional value $v' := \texttt{Some}\, v$, otherwise $v' := \texttt{None}$.

Orchard then normalizes the program to ensure that all variable names are unique, and that each variable is either the result of a bag operation or the result of a release mechanism

(`lap` or `em`). To achieve this, Orchard freshens all variable names, and performs aggressive inlining to eliminate all other variables. Conversely, if a bag operation was originally part of an expression and did not have a name, it is given one. In the resulting normal form, programs make explicit relations between the input database, the intermediate bags and released values, and the output of the program.

Next, Orchard infers dependencies between variables by building a graph with a vertex for each unique program variable. Two vertices (u, v) are connected with a directed and labeled edge f if v is the result of running the bag operation f over u. Since the normalized program only contains two simple bag operations, the label f is either the map function supplied to some `bmap`, or the clip bound supplied to some `bsum`. Since Fuzz forbids unbounded loops over private data, this graph is acyclic. Furthermore, since both `bmap` and `bsum` take one bag variable as input and produce another bag variable as output, there is at most one edge between any two vertices in this graph. This implies the graph is in fact a directed tree, and at the root of this tree is the input bag.

This tree is a complete snapshot of the red zone computations encoded in the normalized Fuzz program. Since the dependency tree tells us how to compute any bag value given the bag variable name, we only need to keep bag variable names at their use sites. So we remove all bag operations from the normalized Fuzz program, and use the dependency tree as a reference for emitting code when a bag variable is used. We call the remaining normalized program the "core".

The core contains a mixture of orange zone and green zone computations. Since Orchard eliminates all other program variables in an earlier pass, the variables in the core must either be the result of a bag computation, or the result of a release mechanism. In particular, we call the variables that are results of bag computations "exit vertices" in the tree. (These vertices are scalar numbers, and thus cannot contain any outgoing edges, because no bag operations take scalar numbers as inputs.) By analyzing the core and inspecting the path from the input database to exit vertices, we can emit code in the `bmcs` form.

5.3.4 Transformation to `bmcs` *form*

The next step traverses the core in a forward pass, while maintaining a intermediate set S of variables. The set S is the set of variables that are results of release mechanisms at the current program position during the forward pass.

When the traversal encounters a release mechanism (`lap` or `em`), it first compares the set of variables used in this release mechanism against S. If the set of used variables is a subset of S, then this release mechanism only adds further noise to already released data, and there is no need to invoke `bmcs`.

On the other hand, if a variable v is used in the release mechanism but is is not a member of S, then v must be the result of some bag operation. In this case, we must invoke `bmcs` to compute v and release.

Let p be the path from the input database to the variable v. Orchard now computes a map function m_p and a clip value c_p as follows. It initializes $m_p := id$ and $c_p := \infty$, then it traverses p starting from the input database. When it encounters a `bmap f`, it updates $m_p := m_p \circ f$; and when it encounters a `bsum c`, it updates $c_p := c$.

In general, a release mechanism may refer to multiple variables $v_1, \ldots, v_i$ that are results of bag operations. For each v_i, Orchard walks its corresponding path p_i to compute m_{p_i} and c_{p_i}. It then fuses these map functions and clip bounds into a new map function $m\ db = (m_{p_1}\ db, \ldots, m_{p_i}\ db)$ and a new clip bound $c = c_{p_1} +\!+ \ldots +\!+\ c_{p_i}$, where $+\!+$ represents vector concatenation.

Finally, if $f(v_1, \ldots, v_i)$ is the release mechanism that uses program variables $v_1, \ldots, v_i$, we build the release function $r\ sum = f(prj_1\ sum, \ldots, prj_i\ sum)$. Here, sum is the aggregated vector, and each prj_i projects the corresponding value for v_i out of the aggregated vector sum.

5.3.5 Optimizations

The transformation process that has been described so far will calculate the correct result, but in general it will produce many redundant `bmcs` operations because it walks the core in a forward pass and emits one `bmcs` call for each release mechanism that uses private data. We can do better by observing that release mechanism calls often do not depend on each other (such as the three calls to `noise` in the k-means example) and can in fact be fused into one `bmcs` call.

Orchard exposes these optimization opportunities to the code transformation process through a simple source code rewriting step. After Orchard has inlined and normalized the input Fuzz program, but before code transformation into `bmcs`, Orchard performs local dependency analysis on release mechanism calls, using a marker combinator `par` to combine release mechanisms that have no dependency relations.

For example, the three `lap` calls in the `noise` function for the kmeans example will be rewritten into:

```
((x', y'), size') ←
  par (par (lap 1.0 x) (lap 1.0 y))
      (lap 1.0 size)
```

Since Orchard inlines the `noise` function, in fact all nine `lap` calls in the `step` function for the k-means example will be combined through the marker `par` combinator (there are three `lap` calls in each `noise` call, and there are three `noise` calls).

The purpose of the `par` combinator is to allow code transformation to fuse release mechanisms together just by looking at the syntax of the program under analysis. In the last phase of code transformation, when Orchard encounters a `par` combinator, it first recursively emits the map and release functions for the two arguments to `par`. Let us call these map functions m_1 and m_2, and the release functions r_1 and r_2. Next, Orchard fuses them together by creating a new map function $m\ db = (m_1\ db, m_2\ db)$, and a new release function $r\ sum = (r_1\ sum, r2\ sum)$. The clip bounds are concatenated to produce a fused clip

bound. The code transformation recursively fuses the release mechanisms combined with nested `par` combinators, until finally only a single `bmcs` call is emitted for all of the combined release mechanisms.

5.3.6 Limitations

Our implementation currently insists that all loops in the red and orange zones terminate after a finite number of rounds, and it disallows unbounded recursion in these zones. Finite loop bounds are common in the differential privacy literature because they simplify the reasoning about the privacy cost; queries with unbounded loops, such as the PrivTree algorithm [304], tend to require more sophisticated reasoning, and thus cannot be verified by most automatic checkers. If necessary, the limit in the red zone could be replaced with timeouts and default values [144]. Notice that we *do* allow unbounded loops in the green zone, so we can still use dynamic predicates to check for convergence, e.g., in k-means clustering.

Orchard's front end relies on an existing programming language and type system, and it inherits their limitations. In particular, if a query is differentially private but the Fuzz type system cannot prove it, Orchard will reject it, and if a query's real sensitivity is s_1 but Fuzz only derives a sensitivity value $s_2 > s_1$, Orchard will use s_2. These limitations could be removed by using a different source language – e.g., one with a more advanced type system, such as DFuzz [123], or one that allows the analyst to help with the privacy proofs, such as apRHL [13].

Orchard's optimization for fusing independent release mechanisms only recognizes fusion opportunities for release mechanisms that are syntactically next to each other. Due to this simplistic nature, Orchard may miss opportunities for fusion of release mechanisms that are only revealed through a more global dependency analysis. However, in our experiments, we find that this limitation does not prevent us from emitting code with the optimal number of `bmcs` calls. We plan on improving the fusion analysis in future work.

5.4. Query execution

Next, we describe the platform Orchard uses to execute distributed queries once they have been transformed using the method from the previous section.

5.4.1 Overall workflow

Orchard implements `bmcs` using the CaT primitive from Honeycrisp, with three important additions: Orchard supports more than one round, it adds the broadcast step (which was not needed for Honeycrisp's one hard-coded query), and it supports more general computations on the user devices and within the committee's MPC (which Orchard needs for the red and orange zones). Protocols for sortition and verifiable aggregation (discussed below) are used verbatim, so the correctness proofs from Honeycrisp still apply. The platform consists of two components: a *server*, which runs in the aggregator's data center, and a *client*, which runs on each user's device (e.g., phone or laptop). These components operate as follows.

Setup: When an analyst wants to ask a query, she formulates it in the language from Section 5.2 and submits it to the server. The server typechecks the query, to verify that it is differentially private; if not, it aborts. The server then transforms the query as described in Section 5.3, but keeps only the code for the green zone. The server then triggers a *sortition* protocol that causes a very small, random *committee* of user devices to be elected. (As in Honeycrisp, a typical committee size is about 30–40, out of perhaps 10^9 devices.) The server sends the query to the committee, whose members perform the same transformation as the server but keep only the code for the orange zone of each `bmcs` operation, as well as the associated privacy costs ϵ_i. The committee runs an MPC to generate a keypair for an additively homomorphic cryptosystem, and each committee member keeps a share of the private key. The server then executes the prefix (if any) of the green-zone computation that does not involve private data.

Broadcast: When the server encounters the ith `bmcs` operation, it sends the sequence number i to the committee. The committee deducts ϵ_i from the privacy budget ϵ^{max} and, if

this succeeds, signs an *execution certificate* that contains the query, the public key, and the sequence number i of the `bmcs`, and returns the certificate to the server. This certificate is needed to convince the clients that the server has "paid" the privacy cost ϵ_i for the specific step they are about to execute; the sequence number prevents query reexecution without charging the privacy budget again.

Map and clip: The server now distributes the certificate, along with any broadcast state in the `bmcs`, to the clients. Each client (1) verifies that the committee was elected properly, that the execution certificate is signed by the committee, and that the certificate is not a duplicate; (2) transforms the query to obtain the red-zone computation for the i^{th} `bmcs` operation; (3) executes the red-zone code on its local data; (4) encrypts the result with the public key from the certificate; and (5) uploads the result to the server, along with a zero-knowledge proof that (a) the local input was in the correct range; (b) the red zone was executed correctly; and, if $i > 1$, that (c) the client has not changed its local input since the first `bmcs` in the current query.

Sum: The server aggregates all the uploads using homomorphic addition and then publishes a Honeycrisp-style summation tree, so the clients can verify that it has included each user's data exactly once; if not, they can report the aggregator. Next, the committee performs another MPC to execute the orange-zone code (which noises and decrypts the computed aggregate) and then sends the plain-text result to the server, which uses it as the result of the `bmcs` operation and continues executing the green-zone code. If the server encounters further `bmcs` operations, it repeats the broadcast, map, clip, and sum steps for each of them.

5.4.2 Security: Aggregator

One key difference from Honeycrisp is that Orchard's red- and orange-zone computations are not hard-coded and must be compiled from the query instead. A naïve approach could have been to have only the server perform the transformation and to have it provide the red- and orange-zone code to the committee and to the clients, respectively. However, in this case it would have been easy for the server to, say, replace the orange zone with the

identity function (to disable noising) and/or to replace the red zone with "if the user is Alice, return data $\times 10^9$, else 0" (without proper clipping).

Orchard avoids this issue by (1) having the committee and the clients compile the red and orange zones directly from the original query and by (2) including the query in the execution certificate, so that all correct participants can be sure they are part of the *same* query. Since a correct client or committee member would perform the compilation as specified, it would (correctly) reject any proposed query that was not differentially private, and it would include all the necessary elements, such as clipping and noising. A dishonest server still has control over the green zone and can run any arbitrary code there. However, it can only hurt itself by doing this: the users' privacy is guaranteed by the red and orange zones, and any data that reaches the green zone is already properly declassified.

Of course, the aggregator can misbehave in several other ways, but the compilation attack is the only one that is specific to Orchard; the others were already possible in Honeycrisp, and the defenses from Honeycrisp continue to apply. For completeness, we briefly review some key defenses below:

Privacy budget: A malicious aggregator could try to run more queries than the privacy budget allows. To prevent this, the budget balance is maintained by the committee. In each round, the committee checks whether the remaining privacy budget is sufficient to execute the query; if so, it signs a query authorization certificate that includes, among other things, the remaining budget and the current round number. This certificate is sent to all user devices, which check it before uploading their responses. If the committee changes, the new members rely on the budget from the previous round's certificate.

Targeting individuals: A malicious aggregator could try to learn the private data of specific users by performing the aggregation incorrectly – perhaps by leaving out data from certain users, or by multiplying the encrypted data from other users with a large constant (which is possible in an additively homomorphic cryptosystem), or even by pretending that

a single user's data is the result of the entire aggregation. To prevent this, Orchard requires the aggregator to construct a *summation tree* to prove that it has computed the aggregation correctly. Each user device checks a small portion of this tree.

Reporting channel: We assume that there is an external channel that devices can use to report the aggregator, if they should discover that the aggregator has misbehaved. Like Honeycrisp, Orchard produces evidence that the devices can use to substantiate such a report; for instance, this evidence could be posted in an online forum (Twitter, Wikipedia, ...) or it could be given to the press. In a large-scale deployment, the aggregator would typically be a large entity with a reputation to lose, so this mechanism should provide an incentive for the aggregator to follow the protocol correctly.

Collusion: If the aggregator is also the manufacturer of the user devices (which would be the case, e.g., in a deployment by Apple or Google), a malicious aggregator could try to roll out a backdoored OS version or manufacture a large number of additional devices, with which it could then collude. Here, our assumption that the aggregator is Byzantine only *occasionally* (the OB in our OB+MC assumption) is critical, because it limits the potential impact of such misbehavior.

Committee tampering: For a committee of size C, Orchard requires that $\frac{2C}{5}$ committee members are honest. With 2–3% Byzantine users, as we have assumed in Section 4.1, the chances of randomly sampling a committee with too many Byzantine users are miniscule; with $C = 40$, the chances of ever encountering it during a period of ten years, with one round every day, would be about 0.001%. However, a malicious aggregator could try to increase this probability by preventing honest users from participating in the sortition. To defend against this, the aggregator must maintain a Merkle tree of all the users, so that the results of the election are verifiable by all devices.

5.4.3 Security: Malicious clients

Another key difference from Honeycrisp is that there can be more than one `bmcs` invocation and that clients can potentially learn some information about the result of previous invocations from the broadcast step. This is not a privacy issue because the type system ensures that any broadcast state has been properly noised, but a group of malicious clients could potentially use this information in a targeted attack.

As a concrete example, suppose a large online retailer uses the k-means algorithm from Figure 8 to calculate the positions for k new shipping centers, based on the locations of their current customers; suppose, further, that a small group of users wishes to ensure that one of the centers is built in their home town. Notice that each `bmcs` broadcasts the set of centroids from the previous round. In the last round, the attackers can use this information to calculate exactly (modulo noise) what their locations would need to be to move the nearest centroid to their town and then change their inputs accordingly.

To prevent adaptive attacks like this, Orchard can optionally use verifiable computation (VC) [238] on the client side. When this is enabled, clients must upload a cryptographic commitment to their local data along with their first `bmcs` response, and they must include, with each response, a zero-knowledge proof that (a) they have executed the red-zone code correctly and (b) their initial commitment opens to the input they used in the current round. With this defense, the attackers can only choose their initial inputs. As we will show in Section 5.6.3, this makes a successful attack much harder.

5.4.4 Handling churn

A third difference is that Orchard computations with multiple `bmcs` rounds can take much longer than Honeycrisp's single-round computation. This raises two concerns: (1) the workload of the committee is somewhat higher, and (2) devices are more likely to go offline during the computation.

To address the first concern, Orchard can optionally choose a fresh committee after a few

`bmcs` rounds. This requires a few more devices to serve on committees, and it adds a bit more work for the overall system because each new committee has to generate a fresh keypair, but it is safe, and it limits the work that any given committee member has to perform. To address churn in the committee, Orchard uses Shamir secret sharing to ensure that the committee can reconstruct the private key even if it has lost a few of the shares because the corresponding committee members have gone offline.

This leaves the concern that some *user* devices will leave (and others join) between rounds. This does not affect correctness, since the red zone retains no state between rounds, but it does mean that the `bmcs` sums could be computed over data from slightly different sets of users. Almost by definition, differential privacy cannot release anything that is specific to particular users, so the overall impact of individual user arrivals or departures should be small [106, §2.3.2]. The effect of higher levels of churn depends on the algorithm and on the kinds of users that are joining or leaving. For instance, consider the effect that a major power outage in a large geographic region – say, the 2003 blackout in the Northeastern U.S. [117] – would have on a query that was already in progress. If the query was choosing facility locations within the United States, the results would be severely distorted, since it would suddenly appear as if there were no users in the Northeast at all. If, however, the query was measuring the age distribution of the users, the impact would be small, since the age distribution in the Northeast would be roughly comparable to the age distribution elsewhere.

5.5. Implementation

For our experiments, we built a prototype of Orchard. We used Haskell to implement the Fuzz frontend and the transformations, and Python for the backend. Our prototype generates and runs the actual red-zone and orange-zone code; for the aggregation (which would be done with millions of users in a real deployment), we benchmark the individual steps and then extrapolate the cost. Overall, our prototype consists of about 10,000 lines of code, and is publicly available [233].

Encryption: For additively homomorphic encryption, we use the Ring-LWE scheme [200]. This works over a polynomial ring $R_p \stackrel{\text{def}}{=} \mathbb{Z}_p[x]/(x^n+1)$, where p is a prime and n is a power of 2. The secret key is a random polynomial $s(x) \in R_p$, and the public key is a pair generated by sampling a random $a \in R_p$ and setting the public key to be $(a, b) \in R_p^2$, where $b \stackrel{\text{def}}{=} a \cdot s + e \in R_p$, for some "error" $e \in R_p$ chosen from an appropriate error distribution. The plaintext space is $\mathbb{Z}_q^l$, where $q, l \in \mathbb{Z}$, $l \leq n$, $q \ll p$ and $|p \mod q| \ll q$. To encrypt a vector $z \in \mathbb{Z}_q^l$, the encryptor generates a random $r \in R_p$, and computes the ciphertext $(u, v) \stackrel{\text{def}}{=} (a \cdot r + e_1, b \cdot r + \lfloor p/q \rceil \cdot z) \in R_p^2$. Decryption is then simply $z = \mathit{round}(v - u \cdot s, \lfloor p/q \rceil)/\lfloor p/q \rceil$, where $\mathit{round}(x, y)$ rounds each coefficient of x to the nearest multiple of y. (We assume the errors e, e_1, e_2 are sufficiently small relative to p/q.)

This encryption scheme allows us to represent our key generation and decryption protocols with a small constant number of additions and one multiplication in the polynomial ring. Moreover, it allows us to pack many 'slots' of ciphertexts into one large ciphertext, with almost no additional cost. Given our security parameter choices, this scheme yields up to 4,096 counters, each with a capacity of roughly 50 bits.

MPC: We use the SCALE-MAMBA framework [176] to implement the MPC operations for key generation and for the orange zones (Section 5.4.1). For key generation and decryption we used the same framework of Honeycrisp (Chapter 4). SCALE-MAMBA supports an arbitrary number of parties and is secure in the fully-malicious model. Operations are performed in a finite field modulo a configurable prime p, which allows for the support of both integers and floating points. This is a natural fit for our Ring-LWE encryption scheme, which also requires an integer modulus, and thus no additional modular arithmetic needs to be implemented within the MPC. In Ring-LWE, the additive homomorphism of plaintexts is modulo some integer q, where $|p \mod q| \ll q$; ideally, $p = 1 \mod q$.

Secret sharing: SCALE-MAMBA also supports Shamir secret sharing [270]. We use this to shard the private key among the k committee members in such a way that any subset of $t+1$ members can reconstruct the entire key. At the same time, t dishonest nodes cannot

learn anything about the key, and $t + 1$ honest nodes can detect any errors introduced by dishonest nodes. This enables Orchard to tolerate the loss of a few committee members. We modified the open-source SCALE-MAMBA source code to reconstruct the secret key automatically, if needed, using the remaining shares.

Verifiable computation: We use the zk-SNARK protocol [40] to enable clients to prove, in zero knowledge, that they have done the red-zone computation correctly, with consistent inputs (Section 5.4.3). For benchmarking, we used the implementation from the Pequin toolchain [232].

Security parameters: We use the LWE-estimator tool [198] of Albrecht et al. [15] to obtain concrete parameters that provide sufficient security based on the best known attacks on LWE. We chose dimensionality $n = 4096$, a 128-bit prime p, and a Gaussian error distribution with $\sigma = \frac{\sqrt{2}}{2}$ (which we approximate as the centered binomial distribution with $N = 2$ trials) in each dimension, which gives over 128 bits of security. For the verifiable aggregation, we use the same choices as Honeycrisp, namely SHA-256 hashes and RSA-2048 signatures.

5.6. Evaluation

Our experimental evaluation is designed to answer four high-level questions: (1) How many private queries can Orchard support? (2) How well do Orchard's optimizations work? (3) How effective are Orchard's defenses against malicious clients? And (4) what are the costs of Orchard?

5.6.1 Coverage

To get a sense of how many (private) queries Orchard can support, we did a careful survey of the differential privacy literature to find queries that are plausible candidates for our highly distributed setting. We collected as many different kinds of queries we could find; we excluded only a) queries that were substantially similar to ones we already had (e.g., different variants of computing CDFs), and b) queries where we simply could not imagine the data being distributed across lots of individual devices.

Table 1 (in the Overview section) shows the queries we found, as well as the papers we found them in. We then implemented each query in Fuzz, taking care to write the queries as they were presented in the papers, and not in a way that would be convenient for Orchard (e.g., with computations already grouped the way `bmcs` would require them).

We found that, out of the 17 queries we found, 14 (82%) were accepted by Orchard. The three queries that did not work were PATE [236], IDC [143], and DStress [235]. These queries are not a good fit for our model. DStress operates on graphs, whereas we assume a set of per-user records. IDC is a "template algorithm" with an oracle function U, and good choices for U require functions beyond simple bag operations. PATE requires training private (un-noised) "teacher" models and then training a "student" model with noisy labels provided by the teachers. In our model, only the aggregator could play the role of PATE's teachers, but we do not trust it to see sensitive data in the clear, so we cannot express this algorithm.

Overall, our data suggests that Orchard is able to execute a wide variety of differentially private queries—even though these queries were designed for the centralized model.

5.6.2 Optimizations

A naïve translation of a centralized query typically results in a lot more `bmcs` invocations than necessary. To estimate how much our optimizations can help with this, we compiled each query twice, once with the full transformation and once with optimizations disabled; we then counted the `bmcs` operations in the resulting programs.

Table 2 shows our results. In most cases, our optimizations substantially reduced the number of `bmcs` rounds that were needed. (The exact reduction depends on the parameters.) Since the rounds are done sequentially (the `bmcs` calls in the green-zone code are "blocking"), and since `bmcs` accounts for almost all of a typical query's runtime, this means a much lower processing time.

We manually inspected the optimized code, looking for opportunities to further reduce the

Query	Naïve	Optimized
ID3	$2md$	$m+1$
k-means	$3m$	$m+1$
Perceptron	$2md$	$m+1$
PCA	d^2+d	1
Logistic regression	$d+1$	2
Naïve Bayes	$2d$	2
Neural Network	$2m(d+1)$	$m+1$
Histograms	b	1
k-Medians	$3m$	m
CDF	b	1
Range queries	b	1
Bloom filters	d	1
Count Mean Sketch	d	1
Sparse vector	1	1

Table 2: `bmcs` rounds needed for each query, with and without optimizations. d is the input vector length, m the number of iterations, and b the number of buckets (see Section 5.6.4).

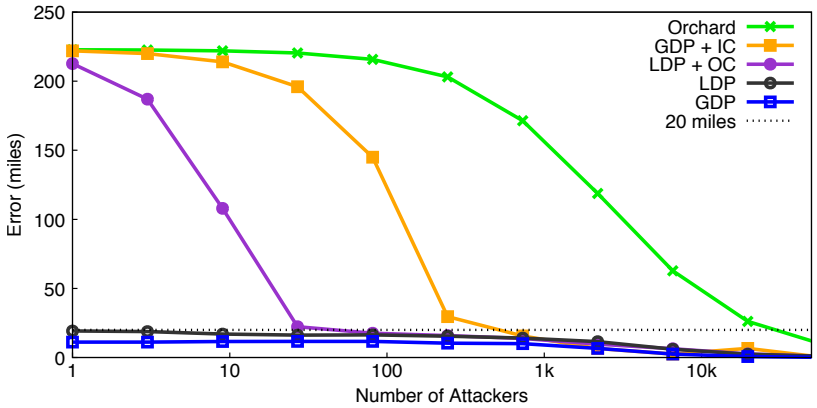


Figure 9: Impact of malicious users.

number of rounds, but could not find any. In principle, Orchard's optimizations could miss opportunities for fusing release mechanisms (Section 5.3.6), but this did not occur for any of the queries we tried.

5.6.3 Robustness to malicious users

To examine how much Orchard's defenses help against malicious users, we implemented the attack scenario from Section 5.4.3. Recall that this involves an online retailer using k-means to find locations for $k = 3$ new shipping centers and a group of attackers trying to cause one of the centers to be built in their home town. We randomly sampled latitudes and longitudes for $N = 10^4$ honest users from a rectangle that includes the lower 48 U.S.

states, and we used Seattle, Houston, and New York as reasonable guesses to initialize the centroid positions. We then simulated the behavior of Orchard, as well as four hypothetical alternatives: (1) local differential privacy (LDP); (2) global differential privacy (GDP) with a trusted aggregator; (3) GDP with input clipping (IC), which rejects coordinates outside the valid range and was implemented in Honeycrisp (Chapter 4); and (4) LDP with output clipping (OC), which requires users to clip their *noised* values to $10\times$ the valid range. The attackers try to move the East Coast centroid (which is near Richmond, VA without the attack) to Pittsburgh, PA, using the strategy from Section 5.4.3; we assume that the attackers do not have knowledge of any data from previous Orchard queries (because, if this information was still relevant, the aggregator would likely have no need to issue a new query). We vary the number of attackers A, and we assume that the attackers are able to estimate N but do not know the locations of the other users. We say that the attack succeeds if the final East Coast centroid is within 20 miles of Pittsburgh.

Figure 9 shows the distance from Pittsburgh of the resulting East Coast centroid for each scenario and with various values for the parameters; the figure shows medians across 500 independent runs. Without a defense, GDP and LDP succumb to even a single attacker, who can observe the centroid's location in the penultimate round and then calculate an input (far outside the valid range) that will move the centroid to Pittsburgh in the final round. The residual error is due to noising; it decreases as A increases. Notice that GDP's error is even lower than LDP's; this is because GDP adds less noise.

With OC, the attackers can no longer report arbitrary values and must instead choose the largest value in the right direction that will be accepted, but the attack still succeeds with about $A = 31$ (0.3% of the users). IC further restricts the range; success now requires $A = 500$ attackers. With Orchard, the attackers cannot adapt, and since they do not know up front what values to report—reporting, say, Portland, ME, would risk "overshooting" and moving the centroid away from Pittsburgh again—their best strategy is to simply report Pittsburgh as their location. With this strategy, the attack takes about $A = 20,000$—far

more than the number of honest users.

5.6.4 Experimental setup

Next, we used our prototype to measure Orchard's costs to users, committee members, and the aggregator. We benchmarked the client-side software on a laptop with a 2.3 GHz dual-core processor and 8 GB of RAM running macOS Catalina. To simulate committee members operating in a global setting, we used `t2.large` EC2 instances with 8 GB of RAM, located in all available geographic regions (including the U.S., Europe, Asia, and Brazil), to get realistic latencies. For our aggregator experiments we used eight PowerEdge R430 servers with 64 GB of RAM, two Xeon E5-2620 CPUs, and 10 Gbps Ethernet; the operating system was Fedora Core 26 with a Linux 4.3.15 kernel. This equipment seems reasonably close to what a real-world aggregator might have available in its data center.

Many of our algorithms have parameters that affect the cost. For k-means and k-medians, we chose $m = 5$ and $k = 3$, because [27] notes that, given proper cluster initialization, the solution after five rounds is consistently as good or better than that found by any other method. For Perceptron, we chose $m = 10$, because the algorithm is guaranteed to converge after at most $O(1/\alpha^2)$ iterations, where α is the margin in a linearly separable dataset [260]. With vectors of size 10, we assume 1-separability to get this guarantee. For ID3, we set vector dimension $d = 100$ because we can support estimating entropy for counters of up to vectors of size 1 million (e.g., all possible 6-digit zip codes) with far fewer counters on the aggregator's side. For the neural network, we chose $m = 20$ epochs, for which [148] shows accuracy competitive with SGD.

Since Orchard is a generalization of Honeycrisp, we report Honeycrisp's numbers for comparison. We got these numbers by executing Honeycrisp's fixed query, which compiles to a single `bmcs`, with Orchard's additions disabled.

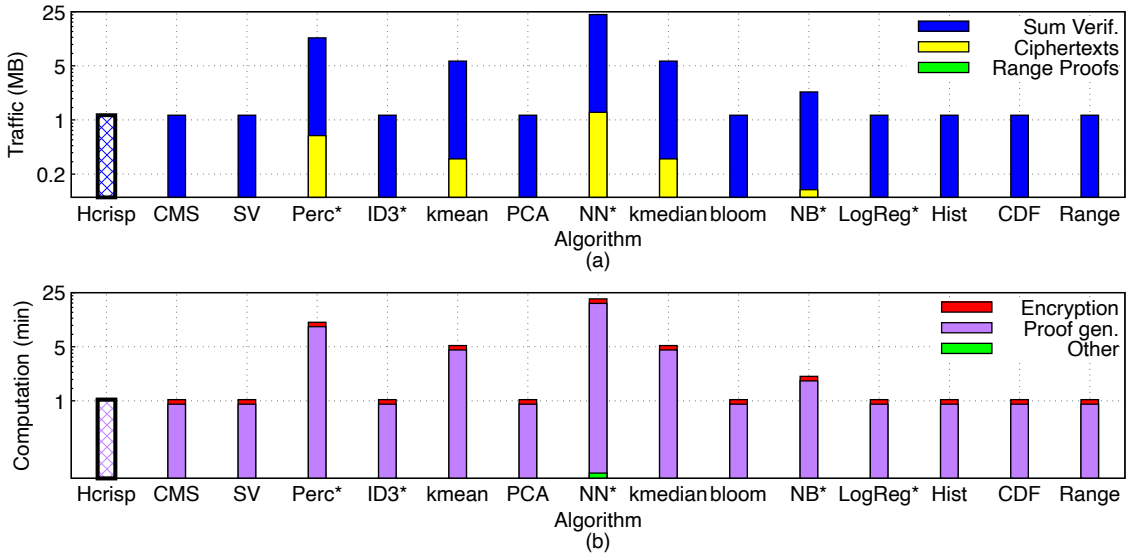


Figure 10: Bandwidth (a) and computation (b) required of each participant in a run of each algorithm.

5.6.5 Cost for normal participants

The key costs to a normal Orchard participant are: (1) the red-zone computation itself; (2) encrypting the value to be uploaded; (3) generating the zero-knowledge proofs; and (4) verifying the aggregator's summation. (The transformations themselves are cheap; this step never took more than 410 ms for any of our 14 queries.) To quantify these costs, we benchmarked the Orchard client while it was executing each of our 14 queries; to get realistic numbers for sum verification, we emulated a system with $N = 1.3 \cdot 10^9$ users for the client to interact with. We measured the number of bytes sent, as well as the computation time spent on Orchard operations.

Figure 10 shows our results. Both the bandwidth and the computation time vary significantly between queries, but they are largely proportional to the number of `bmcs` rounds, whose cryptographic operations dominate the cost. The red-zone computations themselves are typically trivial (many simply return a value), so their cost is very small in comparison; we simply include it with the other protocol overheads in Figure 10(b). Overall, the bandwidth costs are modest, ranging from 1 MB to about 25 MB per query. The computation

typically takes at most a few minutes.

The neural-network query is a an outlier; it takes about 25 minutes of computation time, which raises some concerns, e.g., about battery life on mobile devices. This high cost is mostly due to the high number of rounds we used ($m = 20$), to show what would happen when training on a "hard" problem. For "easy" lower-dimensional problems, even a single pass can be statistically optimal [243].

To measure the cost of the defense from Section 5.4.3, we selectively disabled the part of the zero-knowledge proof that concerns input consistency; this typically reduced the proving time by about 3%. This is because the client already has to prove that the encrypted value is in the correct range; the marginal cost of this extra proof obligation is very small.

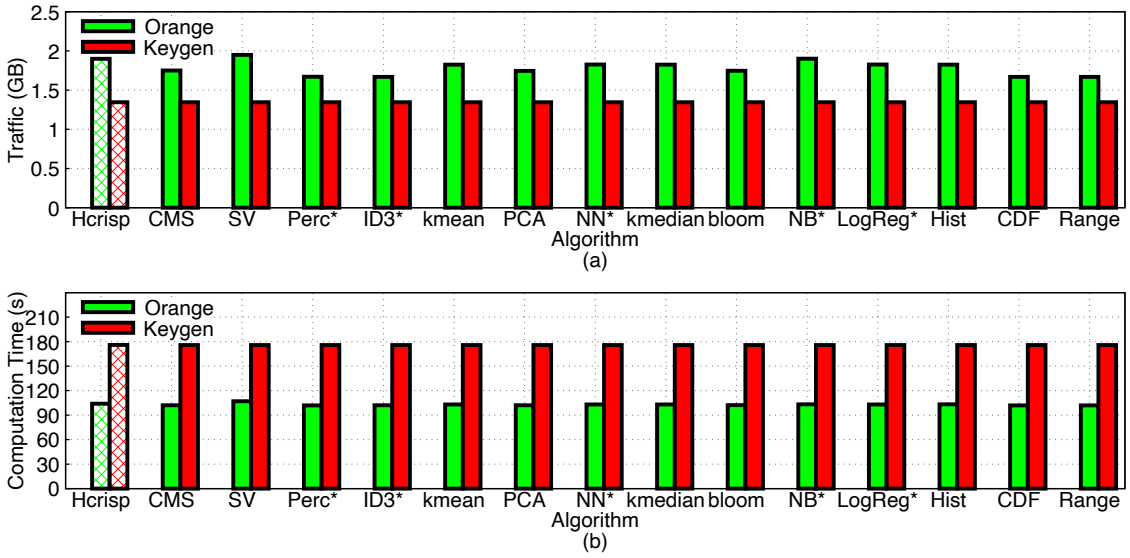


Figure 11: Bandwidth (a) and computation (b) required of each committee member during one round of orange-zone computation.

5.6.6 Cost for the committee

For each query, Orchard selects a small committee of C user devices that are expected to participate in the key-generation MPC, as well as in the per-`bmcs` MPC that performs decryption and orange-zone computations. To quantify the cost to committee members, we set up committees with EC2 instances as described in Section 5.6.4, triggered each of our

14 queries, and measured the bandwidth and computation that the two MPCs consume. We report the cost of a single iteration of each MPC.

Figure 11 shows our results; where queries use two `bmcs` rounds per iteration, we report the cost of the more expensive one (indicated with an asterisk). The cost of the key-generation MPC depends only on the key length, and is thus identical for all queries; the cost of the orange-zone MPC varies with the query, but not by much. Overall, decryption dominates the costs, and, since every `bmcs` call fits into one large packed ciphertext, we see the same behavior for all queries. In absolute terms, these costs are significant; a typical query with one round of `bmcs` consumes about 3 GB of traffic and five minutes of computation time; the total is higher if additional rounds are required.

Notice that the chances of actually being selected for the committee are tiny: for $N = 1.3{\cdot}10^9$ users, a typical committee size is about $C = 40$, so each user is only about $9\times$ more likely to be chosen than to win the jackpot in Powerball. Nevertheless, it may be useful to excuse resource-limited devices, such as mobile phones, from committee service and to rely mostly on devices like desktops and laptops, when possible.

5.6.7 Cost for the aggregator

Next, we quantify the costs of the aggregator, who must collect the input from each device, verify the zero-knowledge proofs, sum up the inputs, generate the summation proof, and distribute this proof to each device. We do not currently have a large enough deployment of Orchard to run this experiment end-to-end, so we estimate the costs based on benchmarks of the individual steps. We set the number of rounds as discussed in Section 5.6.4, and we report results for $N = 1.3 \cdot 10^9$.

Figure 12 shows the number of bytes the aggregator would need to send for each query, as well as the number of Xeon E5-2620 cores it would need to ensure that the computations do not last for more than one hour. As before, the costs depend mostly on the number of rounds; the cost of the green-zone computation is insignificant. The most expensive

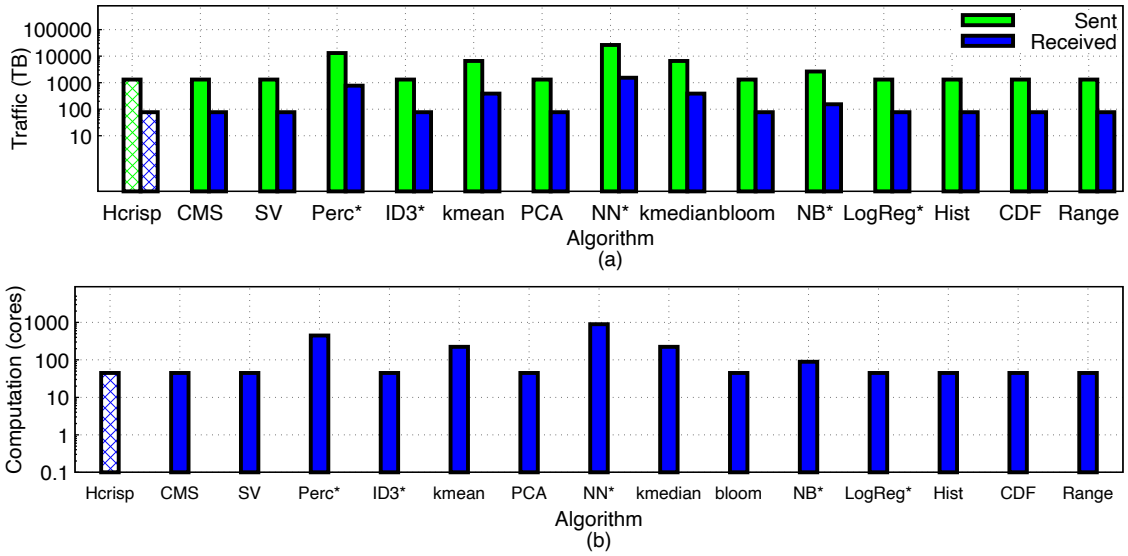


Figure 12: Bandwidth (a) and computation (b) required of the aggregator.

query (Neural Network) would require 892 cores, or 74 machines with two E5-2620 CPUs each. It would also involve sending 13,180 TB, which is a lot but actually corresponds to about 10 MB per user. For comparison: the average transfer size of a web page is about 2 MB [157]; typically, much of this is offloaded to CDNs, and the same would be possible for Orchard's summation proofs.

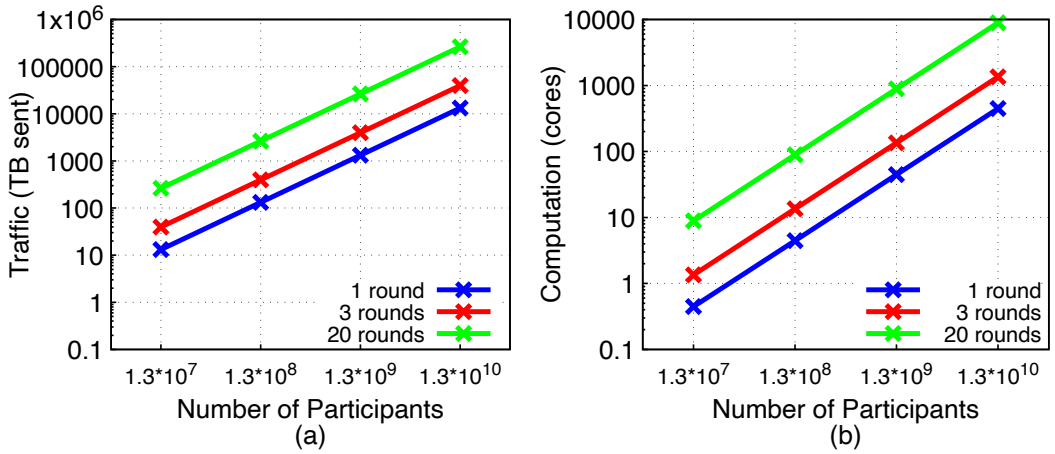


Figure 13: Bandwidth (a) and computation (b) required of the aggregator, for different system sizes.

Scalability: We also ask how well Orchard scales with the number of participating users N. This is mostly a concern for the aggregator: the size of the MPCs (and, thus, the cost

for committee members) does not depend on N at all, and the cost for individual users grows only very slowly, with $O(\log N)$, because of the summation trees. We estimate the costs of the aggregator as above, but this time we vary N.

Figure 13 shows our results (all scales are logarithmic). Although the scaling is technically $O(N \log N)$ because the height of the summation trees grows with N and each user must be sent some paths in the tree for verification, the nonlinear component is small in both figures, which means that Orchard scales very well with N. This is expected, since Orchard is based on Honeycrisp, which scales similarly, and nothing in Orchard destroys this scalability.

5.7. Conclusion

Prior to Orchard, it may have seemed that running differentially private queries at scale required either making compromises (on privacy, accuracy, or trust) or custom-building a cryptographic protocol. Orchard shows that, because of structural similarities among many queries, general solutions do exist, even when there is only a single, untrusted aggregator. There are still types of queries that Orchard does not support—one interesting example are queries on graphs—but we speculate that, by finding and exploiting similar structural patterns, solutions could be built for some of them as well.

CHAPTER 6 : Mycelium

As discussed in previous chapters, personal devices collect massive amounts of data that can enable fascinating applications. For instance, the words typed by smartphone users could be (and in fact are) used to train predictive typing models, which allows phones to offer helpful word completions to users when they are typing.

As another example, the data collected by contact-tracing applications (via Apple and Google's Exposure Notifications API) could be used to understand how diseases spread, or what environmental factors play a role. In this chapter, we describe these settings as instances of *federated analytics* (FA), whereby users, each of whom has a device with some data, collaborate with an aggregator in order to answer questions such as "how often does the word 'system' appear after the word 'operating'?".

Of course, user data—including infection status and demographic information—is very sensitive. Without assurances on who will access their data or what insights will be drawn from it, many users will not comfortably participate in an FA system.

While privacy-preserving FA systems have made considerable progress, including the work demonstrated in Chapters 4 and 5, existing systems lack support for *graph queries* such as: "if a device is infected with malware, how many of their contacts are infected within a week?". This is unfortunate, since graph queries can help study the spread of malware, disease, and misinformation; they could also test for "filter bubbles" and other social phenomena.

However, supporting graph queries privately is challenging due to fundamental differences from the queries traditionally studied in past FA work. In earlier systems, each device analyzes only its local data (e.g., the words that the local user has typed), and the answers are aggregated securely across devices. But in a graph query, each device needs information from *other* devices before it can provide its answer. For instance, in the above example,

even though each user may know their contacts' identities, they would need to find out which ones have been infected. Such an operation raises three technical challenges:

- **Topology privacy**: How can vertices communicate with other vertices without leaking the sensitive topology of the graph to the aggregator? This is especially difficult when the only entity guaranteed to know how to reach all vertices is the aggregator itself (e.g., a user may know the IDs or names of their friends, but not their IP addresses).
- **Neighbor data privacy**: How can vertices collect data from their neighbors and use it to produce their own answer without violating the privacy of their neighbors? For instance, in the above example, how can we prevent users from learning their friends' infection status?
- **Scalability**: How can the system support queries across millions of devices? While it might be possible to build an FA that operates over graphs using secure multi-party computation across devices, these approaches do not scale.

To address these challenges, this chapter introduces *Mycelium*, the first FA system to support queries on massive graphs distributed across a large number of participants. To address scalability, Mycelium's key insight is that, for many graph queries, we can divide the computation into two steps: (1) *local* computations that run in parallel on a small neighborhood of each vertex and output a vector of local results, and (2) a global aggregation step that combines the vertex-level results into a single global output. This is analogous to how frameworks such as Pregel [203] structure their queries, albeit for different reasons. Mycelium cannot support every Pregel query because not all of them are differentially private, but Mycelium's computation model is still quite general.

To guarantee topology privacy, Mycelium needs to provide a way for users' devices to communicate with each other so that they can obtain the inputs needed to execute their local computation (vertex program). This is difficult in many applications without disclosing the existence of the communication to the aggregator. For example, the COVID-19 exposure notification systems use pseudonyms for each device, and there is no obvious way to

communicate with the owner of a pseudonym once it has moved out of Bluetooth range. Mycelium solves this problem by using the aggregator as a rendezvous point, while preventing it from learning the topology of the graph in the process. The key idea is a new mix network and a telescoping circuit mechanism inspired by Tor [96] that allows devices to forward their requests via other devices until the requests reach their destinations (§6.2). To guarantee neighbor data privacy, Mycelium uses homomorphic encryption to aggregate encrypted histograms that are sufficient to answer many queries of interest. We will show several examples of such queries in Figure 15.

A key challenge with Mycelium's mix network is that devices are unlikely to all be simultaneously online, so a fast mixing round could miss some devices—with consequences for both privacy and accuracy. To compensate, Mycelium uses long communication rounds (on the order of hours), so all devices have a chance to contribute their answer; the aggregator buffers messages as needed. Because of the long delays, Mycelium is not suitable for interactive queries; it is intended for longer-term social studies, such as disease spread, investment patterns, or information propagation.

We have implemented a prototype of Mycelium, and we use a combination of small-scale benchmarks and extrapolation to show that it can scale to millions of devices. The cost to the aggregator is well within the means of a typical data center, and the costs to individual devices are moderate: for a typical query, each device will incur around 430 MB of bandwidth and spend 15 minutes of computation. A small, randomly chosen set of devices will need to spend more, but the costs are comparable to what prior FA systems like Honeycrisp and Orchard require at similar scales, even though these systems do not support graphs. In summary, our contributions are:

- A mix network with verifiable telescoping circuits (§6.2);
- Mycelium: the first FA system to support graphs (§6.3);
- A prototype implementation (§6.4) and experimental evaluation (§6.5) of Mycelium.

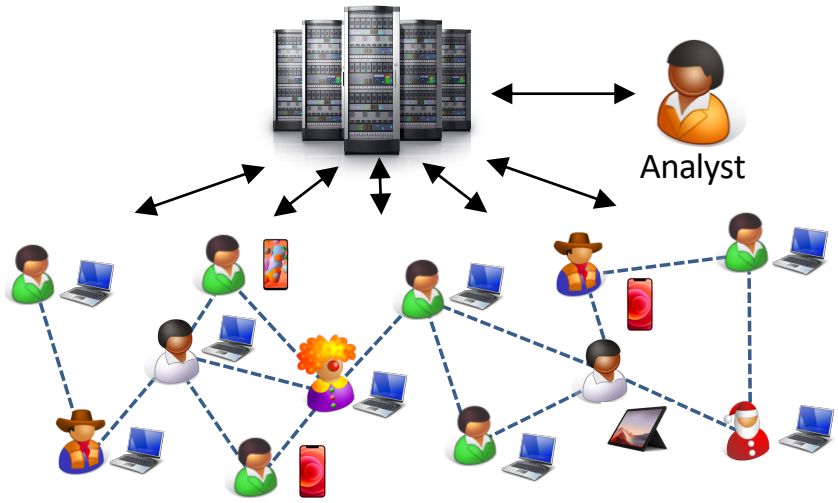


Figure 14: Millions of participants form a graph. An analyst submits queries to an aggregator who facilitates computing on the graph.

6.1. Federated analytics over graphs

We target a setting (illustrated in Figure 14) where there are a large number of *participants*, each of whom has a personal device that contains sensitive information (e.g., financial records, demographic information, health details). Each participant is identified by one or more *pseudonyms*, and participants may know some of the pseudonyms of other participants. For instance, in the case of Google and Apple's Exposure Notification System (GAEN) [1], the devices are users' smartphones; the sensitive information includes users' infection status, time of diagnosis, and locations visited; the pseudonyms could be the Rolling Proximity Identifiers (RPIs), which each phone broadcasts to other nearby phones via Bluetooth Low Energy, or some fixed identifier. Overall, we can think of this data as representing a large graph, with one vertex for each participant and a directed edge (p_1, p_2) whenever p_1 knows at least one of p_2's pseudonyms.

Query	Application	Description
Q1	[9, 246]	Histogram of the number of infections in an infected participant's two-hop neighborhood, within 14 days `SELECT HISTO(COUNT(*)) FROM neigh(2) WHERE dest.inf ∧ self.inf`
Q2	[85, 220, 227]	Histogram of the amount of time A has spent near B, if A is infected within 5-15 days of contact with B `SELECT HISTO(SUM(edge.duration)) FROM neigh(1) WHERE self.inf ∧ (dest.tInf∈[edge.last_contact+5,edge.last_contact+10])`
Q3	[43, 85, 220]	Histogram of the frequency of contact between A and B, if A infected B `SELECT HISTO(SUM(edge.contacts)) FROM neigh(1) WHERE self.inf ∧ dest.tInf ∧ (dest.tInf>self.tInf+2)`
Q4	[43]	Secondary attack rate of infected participants if they travelled on the subway `SELECT HISTO(SUM(dest.inf)) FROM neigh(1) WHERE onSubway(edge.location) ∧ self.inf`
Q5	[220]	Histogram of the number of distinct contacts within the last 24 hours, for different age groups `SELECT HISTO(COUNT(*)) FROM neigh(1) GROUP BY self.age`
Q6	[85, 162, 220]	Histogram of secondary infections caused by participants in different age groups `SELECT HISTO(COUNT(*)) FROM neigh(1) WHERE self.inf ∧ dest.tInf ∧ (dest.tInf>self.tInf+2) GROUP BY self.age`
Q7	[43, 141, 220]	Histogram of secondary infections based on type of exposure (such as family, social, work) `SELECT HISTO(COUNT(*)) FROM neigh(1) WHERE self.inf ∧ dest.tInf ∧ (dest.tInf>self.tInf+2) GROUP BY edge.setting`
Q8	[162, 237]	Secondary attack rates in household vs non-household contacts `SELECT GSUM(SUM(dest.inf)/COUNT(*)) FROM neigh(1) WHERE self.inf GROUP BY isHousehold(edge.location)`
Q9	[184, 220]	Secondary attack rates within case-contact pairs in same vs different age groups `SELECT GSUM(SUM(dest.inf)/COUNT(*)) FROM neigh(1) WHERE dest.age∈[0,100] ∧ self.age∈[dest.age-10,dest.age+10]`
Q10	[162]	Secondary attack rates at different disease stages(incubation vs illness periods) `SELECT GSUM(SUM(dest.inf)/COUNT(*)) FROM neigh(1) WHERE self.inf ∧ (dest.tInf>self.tInf+2) GROUP BY stage(dest.tInf-self.tInf)`

Figure 15: Example queries. `CLIP` commands and histogram bins have been omitted.

There is also a central *aggregator*, who wishes to run large-scale queries over this graph and is willing to coordinate the necessary computation. Note that these queries are not *real-time* queries; at this scale, they may take hours or days to complete. We assume that the aggregator has substantial computational and bandwidth resources, perhaps in the form of a data center. The aggregator works with at least one *analyst*, who formulates the queries to be run. In the case of GAEN, the aggregator could be Google or Apple, or the government agencies that run the Diagnosis Servers; the analysts could be some carefully vetted epidemiologists.

We assume that devices are usually (though not always) online. Devices could be behind NATs or firewalls, or they could go offline for brief periods of time due to loss of cellular coverage or whenever they run out of power.

6.1.1 Example queries

We now provide a few examples of queries that we wish to support. For concreteness, we focus on queries proposed in the infectious disease literature, even though Mycelium is general and can handle graph queries for other domains. Figure 15 summarizes the queries, along with the motivating works, and the corresponding SQL-like syntax.

Superspreading is a well-established phenomenon for infectious diseases [110, 195], and there is work that quantifies the role of superspreaders in pandemics [9, 43, 184, 194]. For example, two works [9, 246] investigate data containing information about chains of transmission or clusters originating from a primary source. Such queries can be formulated as which calculate the number of infected individuals in the N-hop neighborhood of the primary source.

Another line of research analyzes the conditions under which infections most likely occur [43, 85, 141, 162, 184, 220, 237]. In particular, these works calculate secondary attack rates (the probability that an infected individual transmits the disease to an exposed contact) [162, 184] under various conditions. For example, several works [43, 85, 141, 162, 184, 220, 237]

explore secondary attack rates of infected individuals across sex, age, household sizes, and epidemic phases; others [43, 141, 220] explore secondary infections based on exposure type. User devices provide access to location and demographic data, which makes such queries possible. Additionally, with temporal data we can answer queries such as Q2 and Q3.

Right now, these queries are answered through manual tracing; for instance, one study uses data from 391 cases and 1,286 of their close contacts in China [43]. A deployment in a GAEN-like system could potentially provide access to larger data sets. Even in cases where data is collected by a country's public health system [237], privacy concerns still exist [100]. A system like Mycelium would allow queries over sensitive data without violating the privacy of individuals.

Although these queries look different, they are structurally similar: they (1) look at a small "neighborhood" around each vertex in the graph, such as the vertices within two hops; (2) compute something across this neighborhood, such as the number of infections; and finally (3) compute some aggregate statistic about these numbers, such as a histogram.

6.1.2 Threat model and goals

We assume that all parties—the participants, the aggregator, and the analysts—could be potentially malicious (Byzantine). However, following prior work (Chapters 4 and 5) we use the *OB+MC assumption*: we assume (1) the aggregator is honest-but-curious at the beginning and usually thereafter, but could be occasionally Byzantine (OB) for brief periods, and (2) most of the participants are correct (MC), except for perhaps 1–2%. OB basically models a system compromise or an inside attacker who may control the aggregator arbitrarily, but only for a short period of time. If the aggregator were malicious all the time, it could manufacture an unbounded number of colluding Sybils, defeating all known defenses. With 100 million devices, MC still means that there will be 2 *million* Byzantine participants. Our goal is to provide the following properties:

- **Output privacy:** The output of the query should not leak (much) information about

the data of individual users, or about the presence or absence of particular edges.

- **Neighbor data privacy:** The computation that is used to answer the query should not reveal anything about a given user's sensitive data to other users.
- **Topology privacy:** The computation should not reveal the presence or absence of an edge to the aggregator.

Notice that we do *not* try to achieve topology privacy between users; our solution does leak a very small amount of information about the topology to nearby users, which is the presence of multiple paths between two users. This is out of necessity: if we tried to perfectly hide the topology even from nearby users, we could not avoid double-counting data from different pseudonyms of the same user, which would severely limit accuracy. However, users already know, or can know, most of the information that is being leaked, since edges are formed through formal relationships or physical proximity. Another non-goal is that we do *not* try to protect the aggregator from itself: if the aggregator tells lies or otherwise misbehaves during one of its Byzantine periods, it can permanently lose the ability to ask additional queries and would then have to reinitialize the entire system.

In addition to the above three properties, we are interested in solutions that can efficiently scale to millions of participants and do not require additional trusted parties.

For output privacy, we adopt *differential privacy*, as in previous chapters.

In general, differential privacy is difficult to achieve for graph data because graph properties are highly sensitive to changes in vertices and edges. For instance, an undirected linear graph with n vertices has diameter n, but the addition of a single edge between the first and the last vertex cuts the diameter to $\frac{n}{2}$. However, the queries in Table 15 are fairly local; they basically count the vertices whose k-hop neighborhood has a certain property. This type of query tends to have a low sensitivity bound that can be computed statically (§6.3.7).

6.1.3 Strawman solutions

To illustrate the challenges of this scenario, we discuss two strawman solutions.

Plain text. Participants could upload their data and the observed pseudonyms to the aggregator, who could answer queries with standard systems such as GraphX[134] or GraphLab [196]. However, this requires users to trust the aggregator, since it can learn the data and the edges of all users.

MPC. Multi-party computation (MPC) [300] is a way for multiple parties to jointly compute a function on their private data, such that no party learns anything beyond what the output of the function implies. A large MPC between *all* participants that aggregates results and adds noise could achieve our privacy goals, but we are not aware of any MPC that can scale beyond a few hundred participants, whereas our scenario can involve millions.

6.1.4 Our approach

Our key insight is that scalability can be achieved by splitting the computation into two parts: a local part that can be executed by the devices themselves, by exchanging messages with other devices that they share an edge with, and a global part that efficiently aggregates the results of the local part. We adapt Orchard (Chapter 5) for the global aggregation; Mycelium's key contributions are the local computation for graphs, the communication mechanism between devices, and eliminating the need to generate new cryptographic keys for each query. (At the scale of millions of devices, key distribution is a significant source of overhead and complexity.)

Mycelium executes queries as *vertex programs*, analogous to queries in Pregel [203]. Each vertex has some local state, which is initially the private data of the corresponding participant. The computation then proceeds in discrete rounds that each consist of a *communication step* and a *computation step*. In the communication step, each vertex can send a message to each of its direct neighbors in the graph; in the computation step, each vertex can optionally update its state, based on the messages it has received. After a fixed number of rounds, each on the order of an hour, each vertex must set its state to a vector of numbers. These vectors are then summed up in a final *aggregation step*, which also adds the noise that is required for differential privacy and then outputs the final vector of noisy

sums.

The separation into a local and a global part is key to scalability because it preserves the information about the graph topology. Recall from Section 6.1.1 that queries in our scenario typically examine a small local area around each vertex (e.g., the two-hop neighborhood). Thus, the data of each vertex can influence at most a small, constant number of other vertices. If d is an upper bound on this number, and N is the number of devices, we can compute the final result with $O(N \cdot d)$ operations. But if the topology of the graph is encrypted, the information about *which* vertices can influence each other is lost; any vertex could potentially influence any other vertex. Thus, there is no obvious way to avoid operations on all possible pairs of vertices, resulting in $O(N^2)$ operations. With millions of vertices, this can make a difference of several orders of magnitude.

6.2. Communication

In Mycelium's local phase, the devices need to be able to exchange messages with their direct neighbors in the graph, without giving away details of the topology. This is not completely straightforward, because (a) the devices only know their neighbors' pseudonyms, not their identities or IP addresses, and (b) since the devices can be behind firewalls and occasionally go offline, a device and its neighbor may not be able to establish a direct connection, or may never even be online at the same time. We solve this problem using a type of mix network where devices act as mixes and the aggregator acts as an (untrusted) mediator for all messages.

6.2.1 Assumptions and goals

Our goal is a primitive $\textsc{send}((h_1, m_1), \ldots, (h_d, m_d))$ that delivers a set of messages $\{m_1, \ldots, m_d\}$ to the holders of pseudonyms $\{h_1, \ldots, h_d\}$, respectively, with high probability. We make the following assumptions:

1. There is an upper bound d on the degree of each vertex.
2. Devices' clocks are loosely synchronized.
3. Devices have a key pair (pk_i, sk_i) for each pseudonym h_i, and pk_i is linked to the

pseudonym h_i ($h_i = H(pk_i)$).

4. All devices know (a) a tight upper bound, N_D, on the number of devices, and (b) a bound P on the number of pseudonyms that a valid device could have generated within the time period for which a query is valid.
5. There is a public bulletin board (blockchain) that prevents the aggregator from equivocating to the devices.

6.2.2 High-level approach

At a high level, we use onion routing. A device s sends a message m to a pseudonym t, by routing m through a chain of k other devices: s chooses k pseudonyms $h_1, \ldots, h_k$ and then sends $Enc_{sk_1}(Enc_{sk_2}(\ldots, (Enc_{sk_k}(Enc_{sk_t}(m)))))$ to the first hop h_1; h_1 removes a layer of encryption and sends the result to h_2; and so on, until h_k sends the message m to the destination t. If $k > 1$ and at least one device on the chain is honest, the edge between s and t is hidden.

Since devices cannot communicate directly, Mycelium relays messages through the aggregator, who maintains a "mailbox" per pseudonym. This must be done with care: if devices pick up the messages from their mailboxes one at a time, the aggregator could observe that a message deposited by Alice is picked up by Bob, and that Bob then deposits a message in Charlie's mailbox—revealing the chain. We address this by proceeding in discrete rounds and ensuring that each device mixes and forwards different messages in each round. (We call these C-rounds to distinguish them from rounds of the vertex program.) Thus, the aggregator can only observe, say, that Bob picks up several messages, including Alice's, and that Bob then deposits messages in several other mailboxes, including Charlie's. If each device forwards a batch of b messages in each C-round, and there are at most c devices on the chain colluding with the aggregator, then a given message could be in b^{k-c} mailboxes after k C-rounds. For sufficiently large b and k, and small c in expectation, this makes it hard for an adversary to link messages.

The above presupposes that devices are always online, that colluding devices are honest

but curious, and that the aggregator does not drop messages. We discuss how to handle a malicious aggregator later. To handle devices that go offline or drop messages, it is not sufficient to send a single copy of a message to a target as the message may never reach it. To guard against this, each device sends r replicas of each message over different chains. Additionally, to hide the vertex degrees, each device always sends d different messages; if it has fewer than d neighbors in the graph, it sends extra messages to itself, somewhat analogous to Loopix [244].

If every device sends messages to d targets and uses r replicas of each message, the expected batch size is $b = r \cdot d$. Since bigger batches lead to better security, we restrict the choice of hops to a random fraction f of the nodes. This means that when a device is selected as a routing node, it will handle more messages but be selected less frequently. This increases the batch size by a factor $1/f$, without increasing the average workload.

6.2.3 Initialization

To make the above approach work, devices must be able to pick random pseudonyms for building their chains, without giving the aggregator a way to bias the choice towards colluding devices. For this purpose, the aggregator creates a verifiable map M_1 that maps each integer in $[1, N_D \cdot P]$ to a different pseudonym. Since a malicious aggregator could omit pseudonyms or include pseudonyms more than once, it is required to also create a second map M_2 that can be used to audit the first map. This works as follows.

When a new query is issued, the aggregator begins by compiling a list of the P most recent pseudonyms each device has used. It then randomly assigns each device a unique *device number* in the range $[1, N_D]$, and each pseudonym a unique *pseudonym number* in the range $[1, N_D \cdot P]$. Next, it creates M_1 as a binary Merkle hash tree (MHT), whose leaves are of the form (h_i, pk_i, d_i), where h_i is the i-th pseudonym, pk_i is the corresponding public key, and d_i is number of the device that owns the pseudonym. To ensure that the devices have a consistent view, the aggregator then commits to M_1 by posting the root of the MHT to a bulletin board.

Using this information, a device could theoretically look up the n-th pseudonym and its public key by sending n to the aggregator. The aggregator could then take a binary representation of n and walk down M_1's MHT starting from the root, taking a left on level i if the i-th bit of n is zero, and a right otherwise. This would take it to the n-th vertex from the left. The aggregator could then return that vertex's information to the device, along with an inclusion proof (hashes along the path from the leaf to the root), and the device could verify the response by checking that (a) the pseudonym matches the public key, and (b) the path in the inclusion proof matches the path the aggregator should have taken for n. In practice, such a direct lookup would tell the aggregator that the device is using the n-th pseudonym in a chain; we discuss how to fix this in Section 6.2.4.

To enable the devices to audit M_1, the aggregator also prepares another verifiable map M_2, which maps each device number to a leaf $(H(h_1), \ldots, H(h_P), H(pk_1), \ldots, H(pk_P))$, where the h_i are the pseudonyms this device has used and the pk_i are the device's public keys. The root of this tree is posted to the bulletin board as well. Each device then performs two checks using M_1 and M_2. First, it looks up its *own* pseudonyms in M_1 and checks the inclusion proofs. Thus, if the aggregator has omitted an honest device's pseudonyms, that device will detect the problem. Second, each device randomly looks up x pseudonyms, extracts the corresponding device numbers d_i, and asks the aggregator to show that one of the $H(pk_j)$ hashes in the d_i-th leaf of M_2 corresponds to the pseudonym the device has retrieved. If a device submits a lot more than P pseudonyms, this check will fail with high probability, since each of M_2's leaves can hold only P entries; if a device assumes multiple identities, the aggregator will run out of space in M_2, which can have only N_D leaves.

Starting with the posting of the MHT roots, devices use their clocks to mark the fixed length of each C-round.

6.2.4 Path setup

Each device randomly selects r k-hop "paths" for each of the d messages it will send. Recall that the hops should be picked from among a fraction f of the devices.

The devices select each hop i from 1 to k by picking a random number x from $[1, N_D \cdot P]$ such that $(i-1) \cdot f \leq \frac{H(x \,||\, B)}{H_{max}} < i \cdot f$, where H is a cryptographic hash function, H_{max} is the maximum hash value, and B is a random bitstring that is chosen collectively as, e.g., in Honeycrisp (Chapter 4). Notice that at this point the position of each pseudonym in M_1 is fixed, so a malicious adversary cannot bias the selection towards its confederates.

So far, the devices know only the index of their desired hops in M_1. However, they need to know the actual pseudonyms and establish a shared (symmetric) key with each hop. They cannot ask the aggregator for the pseudonyms directly, since this would give away the intended path. Instead, we use a variant of the telescoping scheme from Tor [96], which we describe next (and illustrate in Figure 16). For ease of exposition, we discuss the protocol in terms of a single device and a single path $h_1, \ldots, h_k$, but the steps are done in parallel across all devices' $d \cdot r$ paths.

In the first C-round, the source device s looks up the pseudonym h_1 and public key pk_1 for its first hop by communicating directly with the aggregator. (In this step and all that follow, the response includes both the leaf and the inclusion proof, and the device verifies them in the same way as in Section 6.2.3.) This is safe because the aggregator will be able to observe the connection to the first hop anyway. s then generates a symmetric key $sk_{s \leftrightarrow h_1}$ and a random path id $p_{s \leftrightarrow h_1}$, and uses an authenticated encryption scheme (AE) to encrypt the identity of the next hop, h_2, with $sk_{s \leftrightarrow h_1}$; s encrypts $sk_{s \leftrightarrow h_1}$ using public key encryption ($PEnc$). Finally, s deposits $p_{s \leftrightarrow h_1} || PEnc(pk_1, sk_{s \leftrightarrow h_1}) \, || AE(sk_{s \leftrightarrow h_1}, h_2)$ in h_1's mailbox.

Once all devices have deposited their messages, the aggregator computes (a) a *mailbox MHT* over the messages in each mailbox, and then (b) a *C-round MHT* over all the inner MHTs. It then commits the root of the C-round MHT in the bulletin board, and then proves to each sender that its messages were included in the MHT. This prevents the aggregator from dropping messages without detection. If some devices do not receive the proof from the aggregator, they post a challenge on the bulletin board. If the aggregator did receive messages from these devices, it can respond with the correct proofs; if a challenge is not

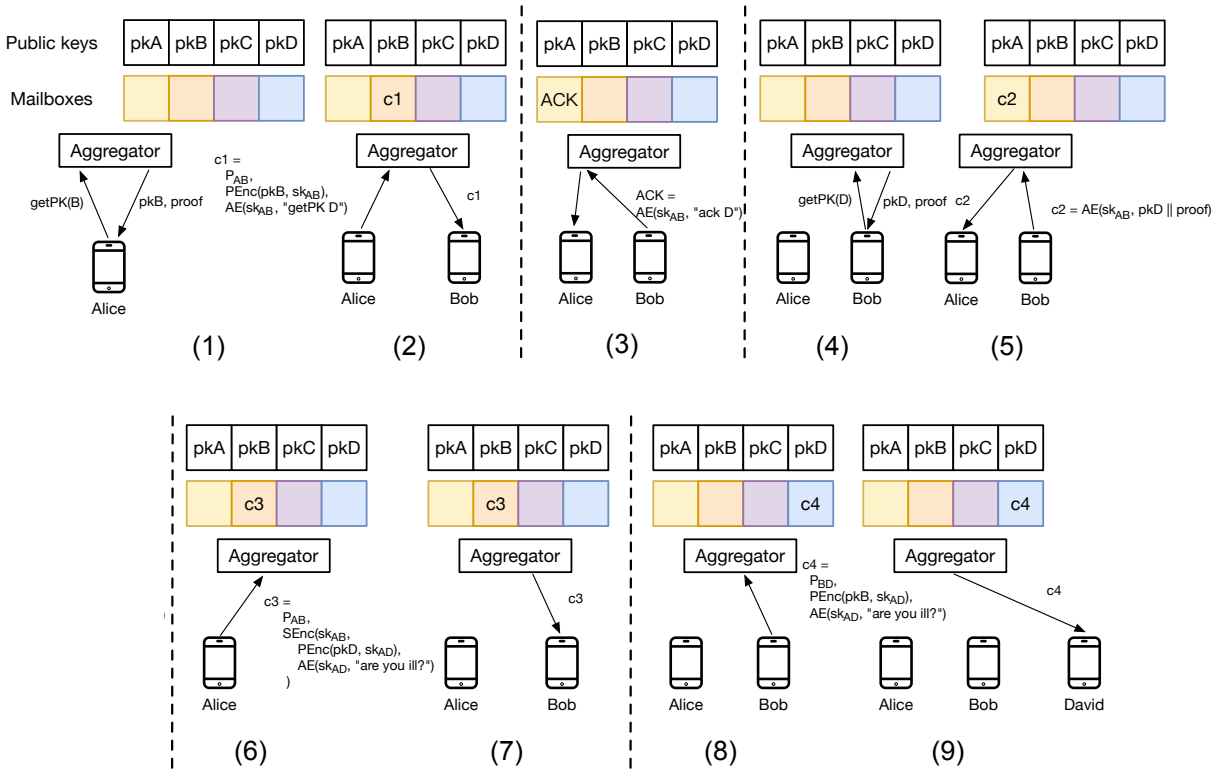


Figure 16: Steps for relaying a first message from Alice to David through Bob (so $k = 1$). Rounds are separated by vertical lines. Steps (1)–(5) correspond to the first 3 rounds and are for path establishment; Steps (6)–(9) are for forwarding. Alice gets the key for pseudonym B (Bob's), directly from the aggregator in Step 1 and then asks B to look up the key for D (David's) in Step 2. After Bob sends an ACK to A (Step 3), it waits k rounds, looks up the key for D, and sends it to A (Steps 4–5). Finally, Alice sends her message along the path (Steps 6–9).

answered, the other devices refuse to proceed, and the path setup has to be restarted without the relevant devices.

Next, h_1 retrieves the batch of messages from its mailbox, and asks the aggregator to reveal the MHT for this mailbox, so it can verify that it has received all the messages. If no misbehavior is detected, h_1 looks up all the public keys for the requested pseudonyms (e.g., h_2) in random order. Then h_1 puts $AE(sk_{s\leftrightarrow h_1}, pk_2)$ in s's mailbox; this corresponds to h_2's public key pk_2 encrypted under the shared symmetric key between s and h_1. At the end of the C-Round, s checks its own mailbox and decrypts the message to learn pk_2.

During the next C-Round, s generates a fresh symmetric key for h_2, $sk_{s\leftrightarrow h_2}$, encrypts

it under pk_2, and sends to h_1: $p_{s\leftrightarrow h_1}||AE(sk_{s\leftrightarrow h_1}, PEnc(pk_2, sk_{s\leftrightarrow h_2})||AE(sk_{s\leftrightarrow h_2}, h_3))$. h_1 then fetches messages from its mailbox and checks the MHT. Finally, h_1 removes the outer layer of encryption of the message from s, generates a new path id $p_{h_1\leftrightarrow h_2}$, locally stores the map $p_{h_1\leftrightarrow h_2}$ to $p_{s\leftrightarrow h_1}$ (to be used in later rounds), and deposits $p_{h_1\leftrightarrow h_2}$ $||PEnc(pk_2, sk_{s\leftrightarrow h_2})||AE(sk_{s\leftrightarrow h_2}, h_3)$ in h_2's mailbox.

This process continues hop by hop: h_2 looks up the key for h_3, h_3 the key for h_4, and so on, until h_k is finally asked to look up the key for the destination *dst*. One issue is that, when h_k receives a batch of requests to fetch the public keys of the final destinations (one of which is *dst*), h_k cannot proceed right away. This is because a malicious penultimate hop (h_{k-1}) could drop the final request sent by s where it tells h_k to fetch *dst*'s key. If h_k were to fetch the keys immediately, the aggregator would observe that *dst*'s public key is fetched fewer times than every other device's, thereby revealing that a device who had h_{k-1} as a penultimate hop had an edge to *dst*—shrinking the anonymity set of *dst*'s edge from $(r \cdot d/f)^k$ to $(r \cdot d/f)^{k-1}$ possible sources. To avoid shrinking the anonymity set, h_k sends an ACK to all sources through the reverse paths, confirming that it has received their requests; if a source doesn't get an ACK, it complains. If h_k does not see any complaint in the bulletin board in k rounds, h_k fetches the public keys in its batch (including *dst*'s). If a source complains, then the last hops in all paths refuse to fetch public keys, and path setup is restarted. On any reverse path to the source, each honest hop knows how many messages it should receive, and aborts if any message is dropped.

At the end, each device knows the pseudonyms and public keys for all the hops along its chosen paths, and it has established a shared symmetric key with each hop. With k hops, this process requires $2+4+6+...+2k+k = k^2+2k$ C-rounds. However, k should normally be small, and the process is run infrequently in order to let new devices join the system. With $k = 3$ and one-hour C-rounds, path setup would take about half a day, which gives flexibility to devices so they can participate even if they briefly go offline.

6.2.5 Message forwarding

Once the paths are set up, communication is as follows. One communication round of the vertex program requires $k + 1$ C-rounds. In the first C-round, the devices onion-encrypt their messages, as described in Section 6.2.2, and deposit them in the mailboxes of their first hops with the path ids generated during the path setup. Then, in each subsequent C-round, each hop downloads the messages from their mailbox, checks to make sure the aggregator did not drop any messages, removes one encryption layer, and mixes them. Finally, they use the mapping generated during path setup to determine the appropriate path id for each message, and upload these with each message to the mailbox of the next hop.

A complication is that the failure of a device could give away some message paths to the aggregator. For instance, suppose that, in a previous round, Charlie downloaded messages from Alice and Bob, and uploaded messages to Doris and Eliot, but in the current round, the message from Alice is missing. If Charlie were to upload a message only to Eliot, the aggregator would be able to conclude that Doris was the next hop after Alice on some path. To counteract this, each hop uploads a dummy message for the next hop for which they do not have a valid message. That way, the communication pattern remains unchanged.

Generating dummies. If messages between a source s and each hop h_i in its path are encrypted with authenticated encryption (AE), then it is infeasible for a forwarding device to generate dummies that decrypt properly owing to AE's existential unforgeability guarantee. This enables the following attack. Let the attacker control h_{i-1} and h_{i+1}. The attacker uses h_{i-1} to drop a message, so h_i generates a random dummy to mask the missing message, and then the attacker uses h_{i+1} to detect which of the messages it received are invalid—thereby learning the relationship between the input and output path ids of h_i. To prevent this, we observe that we only need ciphertext integrity between s and the destination *dst*. Hence, s can construct an onion encryption where the inner ciphertext (the message to *dst*) uses an AE that is indistinguishable from a random message of the same length. For example, encrypt with a stream cipher, then MAC with a PRF, and use the monotonically increasing

round number as a nonce (not sent with the ciphertext to avoid known privacy pitfalls [36]). All other onion layers use a symmetric cipher (*SEnc* in Figure 16) that is indistinguishable from random but that lacks a MAC. This allows h_i to generate a random dummy message which h_{i+1} cannot detect as invalid.

6.3. Query processing

Mycelium evaluates queries in two stages: first, each vertex evaluates a *local* query over its own k-hop neighborhood (say, to compute the number of infected contacts this vertex has), and then the results of the local queries are summed up, noised, and reported to the analyst (say, as a histogram showing what fraction of users have a certain number of infected contacts). In the following, we will refer to the vertex at the "center" of a given local query as the *origin vertex*. Mycelium uses a subset of SQL, with two small extensions, to specify the local queries.

Conceptually, the queries "see" the data as a table `neigh(k)` that contains a row for each member of the k-hop neighborhood, including the origin vertex. The columns of this table are: (a) the private data of the origin vertex (`self`); (b) the private data of the relevant neighbor (`dest`); and (c) the private data associated with the first edge on the path from the origin to the neighbor (`edge`). Queries can ask for `COUNT`s and `SUM`s over columns; we obviously cannot allow direct queries for private data. The `WHERE` predicate can use conjunctions and disjunctions, as well as arbitrary tests within the same column group (e.g., a comparison of two `self` values). It can also contain inequalities over values from different column groups (e.g., `dest.tInf>self.tInf+2`, as in Q3, to test whether a neighbor was diagnosed more than 2 days after the origin vertex) *as long as* both take a finite number of discrete values. Finally, queries can use `GROUP BY` over `self` columns to report statistics for different attribute values.

One extension to SQL is that queries must choose whether the outputs of the local queries should simply be summed up globally (`GSUM`), perhaps to compute a secondary attack rate as in Q8, or aggregated into a histogram (`HISTO`), as in Q1. Another extension is that `GSUM`

queries must specify a "clipping range" $[a, b]$; if the computed value is below or above this range, it is clipped to a or b, respectively.

6.3.1 HE encoding

The two biggest challenges with our protocol are (1) how to implement histograms, and (2) clipping without compromising output privacy or neighbor data privacy.

Suppose, for instance, that we wanted to compute how many users have between 0 and 2, between 3 and 5, and more than 5 infected contacts. Naïvely, we would use private comparisons to implement this: each contact encrypts either 0 or 1, depending on whether they are infected, and sends the ciphertext to the origin vertex, which computes the sum `S` of the values and then uses homomorphic encryption (HE) to compute, say, `IF (0<=S<=2) THEN 1 ELSE 0` for the first bin of the histogram. However, private comparisons between ciphertexts and plaintexts are *extremely* expensive.

Instead, we use the following technique. We rely on the *leveled homomorphic* cryptosystem[1] by Brakerski-Gentry-Vaikuntanathan (BGV) [57], whose plaintexts are polynomials of degree N with integer coefficients, and we encode the value a (e.g., 0 or 1 in the above example) as the polynomial x^a. Then we can use BGV's homomorphic multiplication to add up encoded values: if a device receives $Enc(x^a)$ and $Enc(x^b)$ from two neighbors, it can compute $Enc(x^{a+b}) = Enc(x^a) \cdot Enc(x^b)$. BGV's homomorphic addition then becomes a "bin" aggregation: if one receives $Enc(x^0 + x^1)$ and $Enc(x^0 + x^2)$, then summing these ciphertexts produces $Enc(2x^0 + x^1 + x^2)$, which is an encrypted polynomial where the i-th coefficient gives the number of times that bin i was selected. We can also compute the values in a coarser bin, say $[0, 2]$, by adding up the coefficients of x^0, x^1, and x^2.

The price to pay is that (1) our encoding cannot support more bins than the degree N of the polynomial, (2) the number of local summands cannot exceed the number of multiplications BGV can support, and (3) the number of values to be aggregated cannot exceed the range of the coefficients. This seems fine in our setting: we use $N = 32,768$, which is far larger

[1] A leveled HE supports additions and a small number of multiplications.

than, say, the number of infected friends a given user can have; for reasonable parameters, BGV can support dozens of multiplications; and, with a plaintext modulus of 2^{30}, we can "bin"-aggregate more than a billion values.

6.3.2 Aggregation with Orchard

Orchard (Chapter 5) has the ability to answer a range of non-graph queries, in an otherwise similar setting to ours. The workflow of Orchard also requires a homomorphic encryption scheme, albeit only a simpler *additive* one. Devices encrypt their data and send them to a central aggregator, who sums up ciphertexts. However, the aggregator does not hold the keys for decryption—instead, they are secret shared among a randomly elected *committee* of 10–20 user devices, which use MPC to perform key generation and decryption. The aggregator first uses a summation tree to prove to each device that its data has been included in the sum exactly once; then it sends the aggregate ciphertext to the committee, which decrypts it, adds noise for differential privacy, and returns it back to the aggregator as the final result to the query. This process can be composed over multiple queries, as long as a privacy budget is tracked (see Section 6.3.4).

Mycelium makes two modifications to Orchard. First, it replaces Orchard's additively homomorphic cryptosystem with BGV [57], in order to support both homomorphic additions and multiplications. Second Mycelium observes that, in prior FA systems (including Orchard), each time an analyst wants to run a new query the system must generate and distribute new cryptographic keys to all devices. For systems with millions or billions of devices, such key distribution is both costly and complex. Instead, Mycelium leverages a *verifiable secret redistribution* scheme (VSR) [136] to generate all the cryptographic keys *once*, distribute them to all devices, and then transfer the corresponding private key from one committee to another in such a way that members of different committees cannot collude to recover the key.

In more detail, at the beginning of Mycelium's operation, a set of non-colluding parties, which we call the *genesis committee*, generates all the necessary public keys (including *relin-*

earization keys which the BGV scheme uses to keep ciphertext small after multiplications) and keep secret shares of the corresponding decryption key such that no non-majority of parties can reconstruct the decryption key.

The genesis committee will then transfer ownership of the decryption key shares to the first randomly chosen committee in Mycelium using VSR. Subsequent rounds of Mycelium will likewise perform a VSR transfer of the decryption key from the old committee to a new committee, completely eliminating the need for Orchard's expensive key generation phase. We give more details in Section 6.4.

6.3.3 Basic protocol: Single hop

We first give a protocol where a vertex can answer a query that requires information about its immediate neighbors, and then generalize to a k-hop neighborhood in Section 6.3.4. Processing a query `SUM` over a particular attribute such as `SUM(dest.inf)` consists of the following steps. First, the origin vertex sends a query ID q to all of its neighbors, so they know to which query to respond. Second, each neighbor sends back to the origin vertex a ciphertext $Enc(x^b)$. In the case of a `SUM`, b is the value of the attribute; in the case of a `COUNT`, b is 1 if the predicate applies, and 0 otherwise. After collecting the ciphertexts from each of the neighboring vertices, the origin vertex sums up the encoded values by multiplying the received ciphertexts together, as discussed in Section 6.3.1. The result is a ciphertext of the form $Enc(x^i)$, where i represents the result of the local query over the origin vertex's local neighborhood.

As we discuss later, all of these ciphertexts are then globally aggregated using BGV's additive homomorphism, resulting in a final ciphertext of the form $Enc(\sum_{i=0}^{N-1} c_i x^i)$, where c_i is the number of origin vertices that obtained i as the result of their local query.

6.3.4 Basic protocol: Multiple hops

We now generalize the above protocol to k-hop neighborhoods. For now we assume queries that do not (1) use `GROUP BY`, (2) compute sums over edges, or (3) compare fields from

different column groups. In Table 15, Q1, Q2, and Q4 are of this type. For simplicity, we will assume that the `WHERE` predicate is already in conjunctive normal form.

Flooding. A query over the k-hop neighborhood `neigh(k)` proceeds in $2k$ rounds. As in the single-hop case, in the first round each origin vertex sends a query ID q to its neighbors. In the following $k-1$ rounds, these messages flood to the k-hop neighborhood as follows. When a node receives a message with a given query ID, it remembers from which neighbor it got it. We call this neighbor the *upstream neighbor*. The message from the upstream neighbor is forwarded to all other neighbors. Thus, at the end of the k-th round, each node in the k-hop neighborhood of each origin vertex (a) has received a message from that vertex, (b) knows its upstream neighbor, and (c) knows its distance from the origin vertex, which is simply the number of the round in which the message with a given query ID was first received.

Processing: In the $k+1$-th round, for each upstream neighbor, each vertex evaluates the arguments of each `SUM` or `COUNT` over its local data; for instance, if the query asks for a `SUM(dest.inf)`, each node would look up its infection status, yielding a local result r_i. Next, the vertex evaluates the `dest` clauses of the `WHERE` predicate over its local data; if they all evaluate to `true`, the vertex computes $Enc(x^{r_i})$. If one of the predicates evaluates to `false`, it computes $Enc(x^0)$. Finally, each vertex at distance k from the origin takes each encrypted result and then SENDS it to the relevant upstream neighbor. If a node drops off in the middle of a computation, their value defaults to $Enc(x^0)$, and will thus have a neutral effect on the query's results. From a privacy perspective, this leaks no information about the node's underlying data.

Local aggregation. In round $k-i$, each vertex at distance $k-i$ from the origin receives a ciphertext from each of its neighbors. The vertex evaluates the `dest` clauses and, if they all evaluate to `true`, it multiplies all ciphertexts together, along with an encryption of its own value. The effect is that the vertex now holds an encryption of the *sum* of the encoded values that have been aggregated so far. Finally, unless the vertex is the origin vertex, it

sends the result to its upstream neighbor. If a clause evaluates to `false`, it sends $Enc(x^0)$.

Final processing. In round $2k$, the origin vertex holds a ciphertext which contains the aggregated values over the entire k-hop neighborhood. The origin vertex then evaluates the `self` predicates from the `WHERE` clause; if any evaluate to `false`, it replaces the ciphertext with $Enc(0)$. The origin vertex then contributes the ciphertext for global aggregation.

Global aggregation. The global aggregator receives the ciphertexts from all of the origin vertices and sums them all up. Then, the aggregator gives these ciphertexts to the committee who has the corresponding decryption key (§6.3.2). The committee then decrypts the final ciphertext and adds a calibrated amount of noise based on the query before releasing the result. In particular, let p be the plaintext encoding the underlying aggregated values. For histogram queries, the coefficients of p that fall into each bin of the histogram are summed up, and then, after adding some noise to each bin, the results are released to the analyst. For `GSUM` queries with a clipping range $[a, b]$, the committee clips the range of outputs by computing $\sum_{i=a+1}^{b-1} i \cdot p_i + a \cdot (\sum_{i=0}^{a} p_i) + b \cdot (\sum_{i=b}^{N} p_i)$ and then adds noise and releases the sum to the analyst.

Privacy budget. To bound the privacy loss from multiple queries, the committee maintains a "privacy budget" from which the ϵ cost of each new query is deducted. This is a common approach [102, §3] used in prior FA systems. Our prototype subtracts the full ϵ of each query from the budget, which is safe but conservative. There are several more sophisticated techniques, such as advanced composition theorems [106, §3.5] or sparse-vector techniques (as used in Chapter 4, that would stretch the budget further and that can be used instead.

6.3.5 Special cases

We now discuss how Mycelium handles the special cases excluded in Section 6.3.3. If a query contains a `GROUP BY`, the origin vertex does not just report a single value, but rather one for each possible combination of values in the grouped columns. Our homomorphic

cryptosystem is designed such that all of these values can be packed into a single ciphertext. Only one of these—the one that corresponds to the origin vertex's values in the grouped columns—will represent a non-zero value; the others will be $Enc(0)$. For instance, for Q6, a 20-year old will report a value of 0 for all categories outside of the 18–25 category. Because the parameters of Mycelium support large ciphertexts (§6.4), it can support a fairly large range of possible values in the grouped columns.

If a query compares fields from `self` and `dest` columns, Mycelium does the following. Suppose the comparison is a clause `self.x>dest.y`, and the predicate also contains a `BETWEEN` clause that limits the values of column `y` to a discrete range $[a, b]$. Then, rather than sending back a single ciphertext $Enc(x^m)$, where m is the value in the `y` column, the destination vertex reports a *sequence* of ciphertexts, one for each value in $[a, b]$, with $Enc(x^m)$ in the position corresponding to m, and $Enc(1)$ in all other positions.

During final processing, the origin vertex sums up the subsequence of size ℓ that corresponds to values greater than the value of `self.x`, and then subtracts $Enc(\ell - 1)$ from the sum. This means that the final summed value will be $Enc(1)$ if the destination vertex reported no value (or one outside of the subsequence). Otherwise, the final value will be exactly $Enc(x^m)$. This allows for correct multiplication with the other neighbors' ciphertexts. For example, for a subsequence of length 3, if the neighbor sent $Enc(1)$, $Enc(x^m)$, and $Enc(1)$, the origin vertex will add the ciphertexts received from the neighbor to get $Enc(2 + x^m)$, and then subtract $Enc(3 - 1) = Enc(2)$ from it to get $Enc(2 + x^m) - Enc(2) = Enc(x^m)$.

6.3.6 Malicious nodes

The above protocol returns the correct result if all of the nodes in a device's k-hop neighborhood are correct. But what if some of them are Byzantine? A Byzantine node may not follow the protocol and instead return ciphertexts with coefficients larger than 1, or with more than one nonzero coefficient; the result could be that the aggregator receives a value larger than B. Even if a device itself is correct, it cannot prevent this because it cannot tell what it is computing.

We use zero-knowledge proofs (ZKP) [132] to prevent this attack. When a node sends a ciphertext to its parent, we say that the ciphertext is *well-formed* if it is computed as described above. Each node sends a ZKP to prove to the aggregator that its ciphertext is well-formed. Additionally, each origin vertex sends a ZKP to the aggregator proving that it computed the local aggregation of its k hop neighborhood correctly by multiplying the ciphertexts from its neighbors.

If the ZKP requires a trusted setup (such as Groth16 [140], which we use in our prototype), this setup is performed by the genesis committee (§6.3.2). There are also alternatives that do not require a trusted setup called *transparent zkSNARKs*.

6.3.7 Security analysis

Output privacy. By construction, all queries in our language have bounded sensitivity, and this bound can be statically determined by multiplying the maximum value contribution of any one device by the total number of devices in their local neighborhood. For `GSUM` terms, the max contribution is simply the size of the clipping range; for `HISTO` terms, it is always two because, by changing its local contribution, a vertex can at most decrease the count in one bin by 1 and increase the count in another, also by 1. Thus, we can simply use the Laplace mechanism to achieve differential privacy.

Neighbor data privacy. The message flow is independent of a vertex's private data—in the aggregation phase, each vertex sends back $Enc(0)$ if the `WHERE` predicate evaluates to false—and all the values are encrypted with HE, under a key that neither the aggregator nor individual nodes know.

Topology privacy. The flooding phase reveals to each node (a) the size of its k-hop neighborhood (which is equal to the number of distinct query IDs that arrive), and (b) the number of other node(s) within a k-hop radius that can be reached over more than one directly adjacent edge, and if so, over which edges (because in that case the same query ID arrives over each of these edges). Other than that, the nodes learn nothing about the

topology: they only communicate with their direct neighbors, and the values in the messages they receive have already been aggregated by the neighbors.

Malicious nodes. If a given k-hop neighborhood contains some malicious nodes, these nodes can report incorrect partial sums for their own subtrees of the spanning tree, by encrypting any plausible value (from within $0..B \cdot (d+1)^\ell$, where ℓ is their level in the tree) and computing a matching ZKP, or simply by refusing to send a message to their parent in the tree. However, they cannot cause the aggregator to accept a vector with more than one non-zero coefficient or a vector where the value of the non-zero coefficient is greater than 1 because the aggregator verifies these properties using the ZKP and discards data from nodes whose ZKP is invalid. Thus, a small number of malicious devices cannot have a disproportionate impact on the overall result. We note that discarding invalid inputs introduces a bias towards the data from correct nodes, but (a) the effect should be small, due to the MC assumption, and (b) it seems hard to avoid since there is no way to tell what the *correct input* of a malicious client would have been.

Traffic analysis. Some mixnets, such as Tor, are vulnerable to traffic analysis attacks such as intersection and disclosure attacks [11, 83, 252], in which the adversary observes the traffic in the entire network over some time frame and then makes inferences about whether or not certain participants are communicating. These attacks leverage the fact that mix networks are often *sparse*—that is, only a fraction of participants communicate in any one stage of the protocol. In Mycelium, *every* device participates in *every* mixnet stage, which renders these types of passive attacks infeasible.

6.3.8 Limitations

One obvious limitation of our approach is that there are useful queries that cannot be expressed in our query language. This is not a fundamental limitation—with HE and our communication mechanism from Section 6.2, it should be possible to execute *any* Pregel-like query, as long as the HE scheme supports enough multiplications and the cost of the additional communication rounds is acceptable. The key question is how one would prove

differential privacy. Perhaps a query language such as Fuzz [144] or Duet [224], or even manual privacy proofs using apRHL [13] or CertiPriv [32] could help.

6.4. Implementation

For our prototype, we modified Orchard's codebase in two ways: we (1) replaced Orchard's HE scheme with BGV [57], which required reimplementing the MPC for decryption; decryption; and (2) replaced the ZKPs in Orchard with those of Section 6.3.6. We implemented our mix network from Section 6.2 in C++ using OpenSSL [2] for basic operations (e.g., encryption and decryption). We instantiated `PEnc` using RSA-PKCS1 public key encryption, `SEnc` using ChaCha20, and `AE` using ChaCha20-Poly1305 (nonce is not included in the message). For redistribution of the secret key (§6.3.3), we implemented Extended VSR [136].

Security parameters. For BGV, we set the plaintext modulus to 2^{30}, the ciphertext modulus to a 550-bit prime, and polynomial degree N to 32768. This set of parameters gives over 128 bits of security [14]) and supports 1-hop queries on over a billion users by encoding values of up to 30 bits.

To reduce computation costs on devices, we defer the relinearization for each multiplication to the global aggregation phase, where the aggregator performs a one-time operation to reduce ciphertext size before the decryption step.

MPC and secret sharing. We implemented MPC operations using version 1.7 of SCALE-MAMBA [176], which provides security against up to $\lfloor \frac{k-1}{2} \rfloor$ malicious parties, and performs operations in a finite field modulo a configurable prime p, which helps us support BGV decryption. SCALE-MAMBA also supports Shamir secret sharing [270]; we share the secret key among the k committee members such that any subset of $t+1$ members can reconstruct the secret key, where $t \geq \frac{k}{2}$. At the same time, no t' (where $t' \leq k/2$) dishonest nodes can learn anything about the key, and $t+1$ honest nodes can detect any errors introduced by dishonest nodes. Using the initial setup by the genesis committee (§6.3.2), the secret key is distributed to the first committee. Every committee then uses the extended VSR

[2] https://www.openssl.org

protocol [136] to generate new shares of the secret key for the subsequent committee.

Zero-knowledge proofs. We use ZoKrates [3], a high-level language that can be consumed by SNARK compilers to produce circuits, to express our zkSNARK statements. These in turn can be used with many proof systems, some of which do not need a trusted setup. We use `bellman` [4] as the proof system, which implements the Groth16 scheme [140]. We implemented the proofs for encryption and ciphertext multiplication using this toolbox, and benchmarked the costs for proof size, proving time, and verification time.

6.5. Evaluation

This section addresses four questions: (1) How many queries can Mycelium support? (2) what are the major costs, to normal users, to committee members, and to the aggregator?, (3) how well does the onion routing protect topology privacy?, and (4) how well does Mycelium scale?

6.5.1 Experimental setup

Since we were not able to deploy a system with millions of nodes, we benchmark the various components separately, and extrapolate the costs at scale as done in Orchard. For the client-side and aggregator-side HE benchmarks, we use a MacBook Pro with a 2 GHz quad-core processor and 16 GB of RAM. For our mix net, we run experiments on CloudLab [98] m510 machines with 8-core 2GHz processors and 64 GB of RAM; for the MPC benchmarks, we use 15 Amazon EC2 t2.xlarge instances with 16 GB of RAM. Figure 17 summarizes the parameters we use, unless specified otherwise.

6.5.2 Generality

We first examined the range of queries Mycelium can support. There are two reasons why Mycelium might not support a given query: (1) it is not expressible in the query language from Section 6.3, or (2) the HE scheme in our prototype may not be able to run enough multiplications to process it.

[3] https://github.com/Zokrates/ZoKrates
[4] https://github.com/zkcrypto/bellman

Number of devices	N	$1.1 \cdot 10^9$
Onion routing hops	k	3
Replicas of each message	r	2
Fraction of forwarders	f	0.1
Committee size	C	10
Degree bound	d	10

Figure 17: The parameters we used, unless noted otherwise.

We tried to implement and run each of the queries in Table 15. All queries were expressible, and the query expression is included in the table. This is not surprising because the queries we found in the medical literature compute simple statistics, such as the number of patients for which a particular predicate is true. We were able to run all the queries except Q1. The latter is a two-hop query that would require $d^2 = 100$ multiplications, which exceeds the noise budget of the HE scheme we chose. This is not an inherent limitation; recent HE libraries [5] are close to supporting this number.

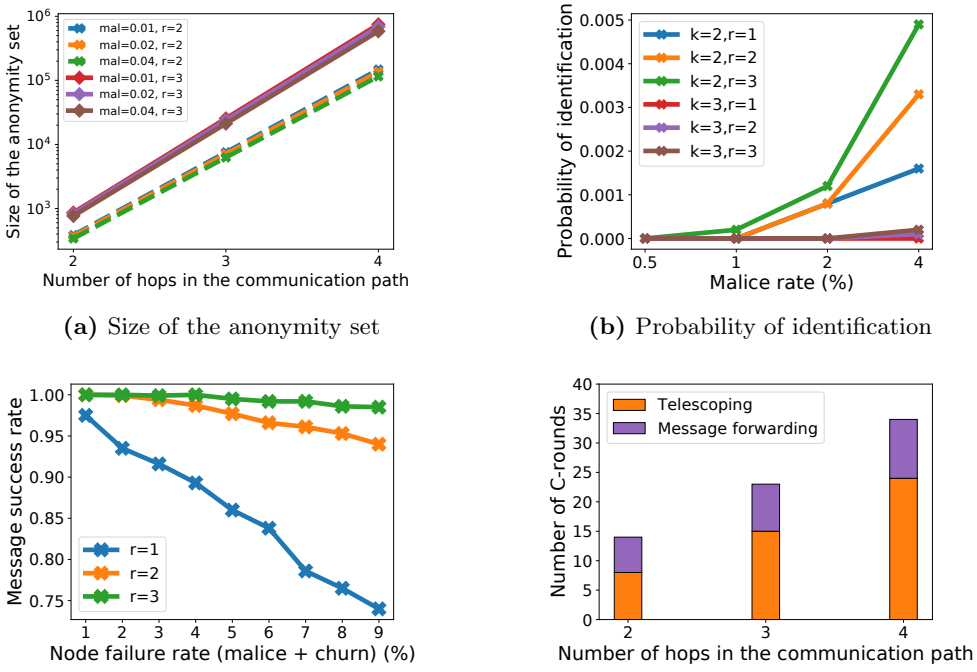
This result suggests that Mycelium can already support many practical queries, which seems encouraging.

6.5.3 Communication layer

Next, we looked at the performance of Mycelium's anonymous communication layer. Recall that Mycelium onion-routes each message on r different k-hop paths, and that, at each hop, each message is mixed with $(r \cdot d)/f$ messages. The aggregator can observe (a) the sets of encrypted messages each forwarder downloads and uploads, and (b) anything that the colluding forwarders saw.

We first focus on topology privacy. Suppose the adversary wants to learn whether there is an edge (a, b). It can observe which messages b downloads at the end, so it can reason about the set of senders that each message could have come from. Each honest forwarder increases the size of this set by r/f—the uploaded message could have been in any of the messages

[5] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL



(a) Size of the anonymity set

(b) Probability of identification

(c) Goodput

(d) Duration

Figure 18: Performance of Mycelium's communication layer

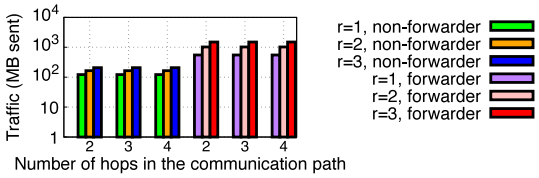
the same forwarder downloaded earlier. Thus, with k honest hops, the number of possible senders is roughly $(r/f)^k$. However, the r replicas of a given message would have come from the *same* sender, so in some cases, the adversary can intersect the r sets. However, because there are more total messages in the system, and the probability of multiple intercepted messages is relatively low, increasing r still (on expectation) leads to larger anonymity sets. Figure 18(a) shows how the expected set size changes with r and k. For our parameters of $r = 2$ and $k = 3$, a malicious fraction of 0.02 still yields an anonymity set of over 7000 devices.

However, a node can be "unlucky" and choose a path that consists only of malicious nodes. In this case, the adversary can identify this exact node as the sender of the message. Figure 18(b) shows the probability for this case. With our default setting of $k = 3$, each query gives the adversary a chance of $p \approx 10^{-4}$ to identify a given edge.

Another concern is that message might not reach its destination because all r copies are

Queries	Number of ciphertexts
Q1, Q2, Q4, Q5, Q8	1
Q3, Q6, Q7, Q10	14
Q9	10

(a) Number of ciphertexts sent for each query



(b) Avg. bandwidth required of each participant per query

Figure 19: User participation costs per query

dropped—either on purpose, by malicious forwarders, or by accident if a forwarder goes offline and does not return by the end of the C-round. Figure 18(c) shows "goodput," the probability that a given message is successfully received (without modifications by adversaries). With $r = 2$ and a node failure rate of 4% (including both malicious nodes and departures), only about one in 100 messages is lost completely. Queries can handle this case, e.g., by specifying a default value for missing inputs in the local aggregation, by counting the number of local aggregations where this (detectable) condition occurs, and/or by asking the local aggregators to upload a final value only when all inputs have been received.

A final question is how long forwarding takes. Figure 18(d) shows the number of C-rounds that are needed for telescoping ($k^2 + 2k$) and forwarding ($2k + 2$, since each query requires a message for the query and a message for the response). If $k = 3$ and C-rounds are one hour long, then both phases of a one-hop query will finish in less than a day. (The duration depends only on the number of hops and not on what specifically the query computes.) This is fine, since Mycelium is not for real-time queries.

6.5.4 What is the cost for normal users?

Next, we examined the bandwidth and computation cost of Mycelium for normal user devices. Each device performs up to three operations: (1) it prepares its own contributions to its neighbors' local aggregations; (2) it potentially acts as a forwarder during onion routing; and (3) it completes a local aggregation for its own neighborhood.

The communication costs vary between queries, depending on how many FHE ciphertexts

they require; each one is around 4.3 MB. Figure 19(a) shows the number (C_N) of ciphertexts for each of the queries in Table 15. In the following, we focus on the cost of a basic query with $C_N = 1$ ciphertext, such as Q5; for the more complex queries, the communication costs need to be multiplied by the number of ciphertexts.

Figure 19(b) shows the communication cost per device. The figure contains two column families: one for the case where the device is selected as a forwarder, and one for the case where it is not. For each case, we vary the number k of hops during onion routing and the number r of copies that are sent of each message. The costs are dominated by message forwarding: each device has to send $r \cdot C_N \cdot d$ large FHE ciphertexts, where C_N is the factor from Figure 19(b), and, when chosen as a forwarder with probability f, it has to download and upload $(r \cdot C_N \cdot d)/f$ of these ciphertexts. For our default parameters from Figure 17 and a simple query with $C_N = 1$, this works out to 1030 MB for forwarders and 170 MB for non-forwarders, or around 430 MB on expectation, given that a $k \cdot f$ proportion of participants will serve as forwarders. For comparison, this is about the cost of sending a four-minute video attachment from an iPhone.

The computation time per device mainly depends on the time to perform ciphertext operations, including encryption and ciphertext multiplication for neighborhood aggregation, as well as the time to generate the ZKPs. The ciphertext operations take around 14 minutes in total with our Python implementation, and the ZKP proof generation takes around a minute, so the total computation time per device is roughly 15 minutes. We implemented an (unoptimized) version of BGV in Python for compatibility with the MPC and ZKP software, so these costs could be dramatically reduced to make use of existing HE optimizations. The computation times for telescoping and message forwarding were negligible, and the costs did not vary much between different queries.

6.5.5 What is the cost for committee members?

For each query, a small committee of C user devices is expected to participate in the decryption MPC using their shares of the secret key. Our EC2 benchmarks show that,

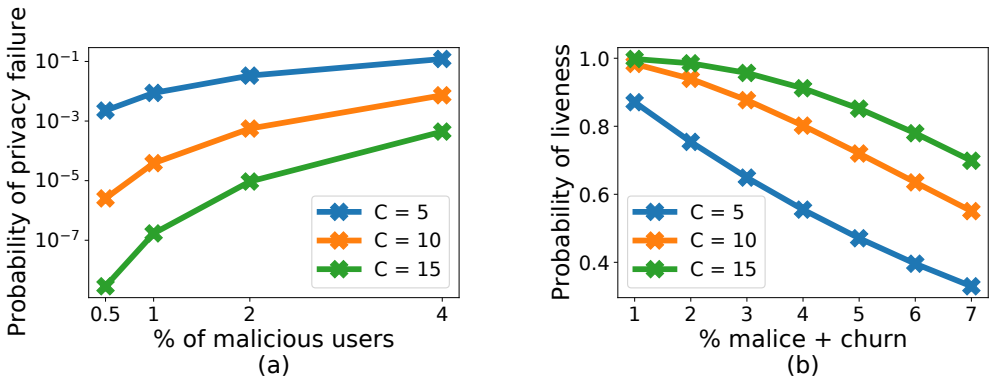


Figure 20: Probability of privacy failure (a) and liveness (b) with different committee sizes.

although Mycelium uses a different cryptosystem, the cost of this MPC is comparable to Orchard's: with a committee of size 10, the total computation time needed was around 3 minutes and the bandwidth required per member is around 4.5GB, plus the (negligible) bandwidth for resharing the secret key. With millions of devices, an individual user's chances of having to serve on a ten-member committee are very small; nevertheless, due to the high bandwidth, it may be best to rely on desktops or laptops where possible.

Figure 20 allows us to reason about the tradeoffs associated with using different committee sizes: a higher committee size provides more security over time (because a larger committee is less likely to contain a majority of malicious members), but also increases the bandwidth and computation time required. Figure 20(a) shows the probability that malicious committee members could reconstruct the secret key, thus causing a privacy failure. In this case, a new trusted setup must construct a new secret key. Figure 20(b) shows the probability of enough committee members being present to decrypt. If there aren't enough members, we simply have to wait until enough are back, and retry the computation.

6.5.6 What are the costs to the aggregator?

Recall that all messages are sent through the aggregator, who maintains mailboxes for each device. Figure 21(a) shows the total amount of traffic the aggregator would need to send to each device, depending on the number of onion-routing hops k and number of replicas per message r. As expected, there is a substantial amount: for our choice of $k = 3$ and $r = 2$,

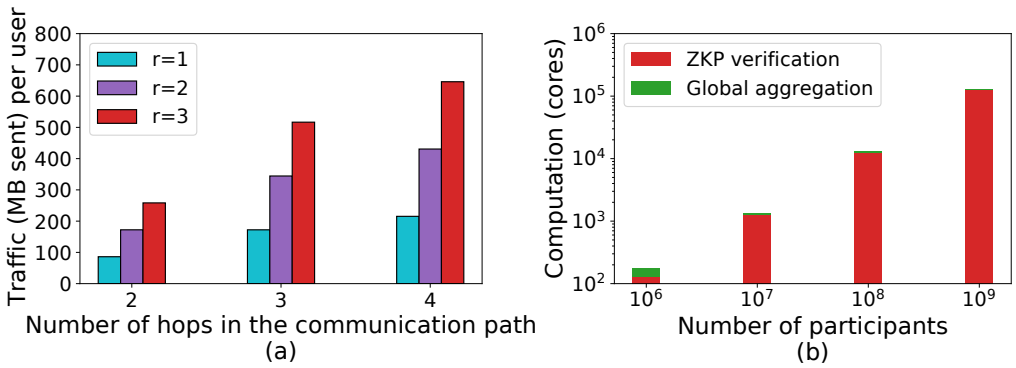


Figure 21: Per-user bandwidth (a) and total computation (b) required of the aggregator for each query.

the aggregator would need about 350 MB per device, or roughly the size of a 10-minute 1080p YouTube video.

The aggregator also needs to verify the ZKPs of each user and perform a global aggregation of ciphertexts. Figure 21(b) shows the number of cores needed to finish the computation within 10 hours with different system sizes. The cost is dominated by the ZKP verification (the bars for the aggregation are very small). Although zkSNARKs normally have small, constant proof sizes, the scheme we use (Groth16) scales linearly in the public I/O size, which, in our case, includes the fairly large ciphertexts. If necessary, the aggregator could reduce this cost by spot-checking only a fraction of the ZKPs, or it could stretch the computation over a longer time.

6.6. Discussion

Cost: It is clear that Mycelium's privacy comes at a high cost—queries on non-sensitive data could be answered cheaply by simply uploading the data to the aggregator in the clear and using a traditional graph-processing system such as GraphX [134]. Indeed, we implemented Q1 for a 1-hop neighborhood in GraphX and ran it on a CloudLab machine with a random billion-node graph and random data. The query finished in about 5 seconds. Mycelium is meant for queries on highly sensitive data that would make the aggregator a target for attacks if it were collected in the clear, and queries that cannot even be asked

today because no single aggregator can be trusted with the necessary data.

Device heterogeneity: In a practical deployment, one challenge would be the wide range of device capabilities. Serving as a communication hop or committee member seems fine for a laptop or workstation that is connected to a wired network, but could be problematic for a mobile phone with a metered cellular connection and limited battery capacity. However, we note that mobile devices are increasingly part of device federations (e.g., a laptop, mobile phone, and smartwatch all sharing the same iCloud account).Since the devices in a federation are typically owned by the same individual, they could safely share their data and designate the most powerful device—say, the laptop—as a participant in Mycelium.

Communication steps could also be delayed when a device is on the road, and resumed when it is plugged in and on a WiFi connection. Finally, hops and committee member selection could be biased towards more powerful devices; this would give the adversary a small advantage, since all of its confederates could claim to be powerful, but one could use slightly more aggressive parameter settings to compensate.

Aggregator workload: For the aggregator, the major costs are communication bandwidth and ZKP verification. Much of the bandwidth is due to the very large HE ciphertexts (4.3 MB), but we speculate that future HE schemes will eventually reduce this cost. For ZKP verification, we note that the 10-hour limit for Figure 21(b) was somewhat arbitrary; in practice, ZKP verification could be done in the background, whenever a data center has spare capacity, as long as the query results are not needed immediately.

6.7. Related Work

Related work on general private analytics at scale has already been discussed in Chapter 3; here we discuss prior work that is specifically relevant to Mycelium.

Private analytics from multiple domains. There is some work on differential privacy which considers aggregating data from multiple domains, like [121]. However, most target relational data: PDDP [69] builds histograms and DJoin [222] computes database joins.

Neither is sufficient to answer graph-based queries. DStress [235] can handle graph data, but does not scale beyond thousands of users. Of the systems that work at scale, including academic works [122] (and prior chapters in this thesis), and deployed solutions [22, 42, 46, 52, 79, 94, 113, 115, 242], none handle graphs.

Traditional graph processing. Graph analytic frameworks [70, 86, 129, 134, 155, 182, 202, 225, 226, 298, 311] target scale but not privacy. Work on social networks has dealt with issues of anonymity [25, 120, 307, 309], but the proposed mechanisms either focus on answering limited differentially private queries [47], on aggregate network estimations that may hide effects of individual malicious nodes [150], or on previous definitions of privacy like k-anonymity [193].

Private contact tracing. Work in contact tracing does not support a single aggregator, or is not designed for central analytics. Mazloom and Gordon [205] support Pregel-like graph queries but require two servers to split trust between them, and does not guarantee differential privacy. Poirot [305] gives differentially private contact summary aggregation, but also splits trust amongst multiple servers, which perform a joint MPC. In the last year we have also seen the design of several other exposure notification and proximity detection systems that give user-level insights [1, 64, 284]. These insights include notifying individuals when they are likely to have been exposed to an infection, but do not support graph analytics.

Anonymous messaging. Mycelium's messaging layer is inspired by Tor [96]. However, Mycelium must operate without the equivalent of Tor relays, and since the devices themselves cannot necessarily communicate directly with each other, it has no choice but to relay communication through the aggregator, which is a global, active adversary. Mycelium's messaging can be seen as a different mix network architecture [17, 68, 80, 178–180, 185, 217, 244, 286, 287] that has high latency and prioritizes privacy over availability, but that has the benefit of not requiring prior pairwise sharing of cryptographic material between senders and the chosen mixes, and balances the load across different sets of mixes

every run of the protocol, which helps the system scale to billions of users.

6.8. Conclusion

Mycelium is the first system to support differentially private analytics on graph queries at a massive scale. It leverages HE, a new mix-network, and Pregel-style queries on top of a Honeycrisp-like architecture. Because ciphertexts must support aggregation of up to a billion devices' information, the costs of Mycelium are higher than similar FA systems. Future work may incorporate cryptographic advances that improve these costs while supporting richer graph queries.

CHAPTER 7 : Arboretum

In this chapter, we continue to consider the scenario where a central entity, the *aggregator*, would like to answer queries about sensitive data that is generated by millions or billions of user devices. The existing options for massive-scale federated analytics, presented in the prior three chapters, still only support a limited set of queries, including mostly numeric queries that can be expressed as sums (Chapters 4, 5), or graph queries with a very specific structure (Chapter 6). This is unfortunate because differential privacy itself supports a much larger range of queries; for instance, categorical queries ("What is the most common zip code?") can be answered using the exponential mechanism [207], and numeric queries can be answered more efficiently using secrecy of the sample [276]. However, we are not aware of any efficient instantiations of these mechanisms for a federated setting with billions of users and thousands of categories; the closest solution we are aware of is an implementation of the exponential mechanism by Böhler et al. [51] that targets deployments several orders of magnitude smaller than ours.

Why are there so few solutions in this space? One reason is that, while more general solutions could *in principle* be built from powerful cryptographic techniques such as fully homomorphic encryption (FHE) or multi-party computation (MPC), *in practice* the necessary bandwidth and computation power, at the scales we are considering, often exceeds even the resources of 800-pound-gorilla aggregators such as Apple or Google. And even when efficient solutions do exist, these solutions often involve subtle optimizations and intricate combinations of different cryptographic techniques that are specific to each particular query. But designing a custom protocol for each new query is difficult and expensive, so this approach is not likely to be practical for most applications.

In this chapter, we present Arboretum, a Federated Analytics system that can solve both of these problems. Arboretum is based on two key insights. The first is that, while the massive number of user devices is certainly a challenge because of the enormous amount of

computation power it requires, *it is also an opportunity* because the devices can help with the computation. True, each individual device (e.g., a cellular phone or a laptop) has very limited resources, but, due to the sheer number of devices, even small contributions from a subset of the devices can add up to an enormous amount of computation power. For instance, a Dell PowerEdge R7525 server has a multi-core Geekbench score of 67,954, whereas a second-generation iPhone SE has only 3,027. However, a billion iPhones computing for one second each can still outperform 10,000 servers computing for an hour! Moreover, leveraging the user devices enables organic scaling: adding devices increases both the demand for resources *and* the supply. This approach has been used successfully, e.g., for CDNs [306] and decentralized systems [158, 259]; as we show here, it also holds promise for Federated Analytics.

Our second insight is that the design space has a fairly regular structure: queries contain high-level operators that can be instantiated in different ways (for instance, sums can be computed with sum trees of different fanouts, with a regular for loop, etc.), and computations can be carried out by different entities and/or with different cryptographic primitives. In our experience, the main sources of complexity are the sheer size of the design space, where even simple queries can be executed in millions of different ways, and the complicated dependencies: for instance, using a slightly slower implementation for one operator can massively speed up another, and using a particular cryptographic primitive might speed up additions but slow down comparisons, which can vastly increase or decrease the overall speed, depending on what else the query is doing. These dependencies are hard to track for a human developer, but they are easy to explore mechanically, using a type of query planner.

We have built a system called Arboretum that accepts queries written in a simple high-level language and then automatically finds a way to execute them efficiently in a federated setting. The queries can be formulated as if all the data existed in a central place, without regard to distribution or confidentiality; Arboretum automatically breaks the computation

into smaller blocks, assigns the blocks to suitable entities for computation (possibly using homomorphic encryption or MPC), and chooses an efficient cryptosystem. We have implemented a prototype of Arboretum, and we have applied it to ten differentially private queries, including six new queries that use the exponential mechanism and/or secrecy of the sample, and four existing queries from Honeycrisp (Chapter 4), Orchard (Chapter 5), and Böhler and Kerschbaum [51]. The results from our experimental evaluation show that 1) Arboretum can efficiently execute all ten queries with billions of participants; and 2) its performance on the existing queries matches that of the earlier systems. To our knowledge, Arboretum is the first system to provide support for the exponential mechanism and for secrecy of the sample in the federated setting, at least at the massive scales we are considering.

Like all query planners, Arboretum is not able to innovate – it can only find solutions that can be derived using the techniques it knows about. Thus, it can most likely be outperformed by hand-optimized solutions and clever new algorithms. However, our results show that Arboretum can find good solutions in seconds or minutes, without a human expert in the loop, and it is easy to extend, e.g., by adding new algorithms or support for additional cryptographic techniques. Our contributions are:

- Arboretum's query planner (Section 7.2);
- a generalized query runtime (Section 7.3);
- a prototype implementation (Section 7.4); and
- an experimental evaluation (Section 7.5).

7.1. Overview

7.1.1 Strawman solutions

FHE only: At first glance, it may seem that FHE solves our problem: the participants could encrypt their data with FHE and upload it to the aggregator, who could evaluate the query on the ciphertexts, add the requisite amount of noise, and decrypt only the final result. However, this approach is impractical for at least two reasons. The first is that, at

the scale we consider here, it can require a gigantic amount of computation. (Especially for the exponential mechanism; see below.) Second, and more importantly, this approach still requires users to trust the aggregator a great deal: if the aggregator can decrypt the final result, it can also decrypt the input data from an individual user. Thus, in terms of privacy, this approach provides very little benefit.

All-to-all MPC: Another possible approach would be to have the the participants input their sensitive data to a large MPC that evaluates the query and returns the final result. This solves the privacy issue from above, but it is even worse in terms of cost: we are not aware of any MPC protocol that can scale beyond a few hundred participants.

MPC committee: A third approach, which is taken by Böhler and Kerschbaum [51], delegates the MPC to a small committee of participant devices. This scales better – at least to a few million participants – but the committee eventually does become a bottleneck: with a billion participants, even downloading the input data from each participant would already generate terabytes of traffic, and evaluating a huge circuit in MPC would add many terabytes more.

HE + MPC committee: The previous chapters in this thesis, Honeycrisp (Chapter 4) and its successors Orchard and Mycelium (Chapters 5 and 6), avoid this bottleneck by having the aggregator sum up the input data, using homomorphic encryption, and by using the MPC committee only for key generation, noising, and decryption. This approach does scale to billions of devices, but so far it has been demonstrated only for a limited range of queries – mostly Laplace-mechanism queries that can be expressed as sums plus some postprocessing. Orchard does support the exponential mechanism but, due to its limited scalability, was only evaluated with ten categories.

7.1.2 Challenges

When one moves beyond Honeycrisp's limited range of queries, the problem becomes substantially more complicated, for several reasons. One is that other mechanisms tend to require far more computation than the Laplace mechanism. For instance, suppose we use

the exponential mechanism to choose a U.S. zip code. In general, this requires evaluating the quality score $q(r, d)$ for *each* possible output r – in this case, the 41,683 possible zip codes. This alone increases the computation cost by more than four orders of magnitude. In addition, the exponential mechanism also requires more expensive *kinds* of computation: for instance, choosing the highest quality score requires comparisons, which cannot be done in AHE alone and thus requires FHE. We estimate that, with a naïve TFHE-based implementation, an aggregator with 10,000 cores would need 3.7 *hours* to compute the quality score for a single output across a billion users; handling all 41,683 possible outputs would require a 400-trillion-gate circuit, which would take more than 17 years to evaluate.

Of course, optimizations can often bring the cost back down to a practical level – but there are lots of different optimizations that make sense in different parts of the design space, and these optimizations interact in complicated ways that are hard for a human developer to track. As an illustration, we show a small part of the design space for one of our queries (`top1`) in Figure 22. (The full space, with two parameters and six metrics, has eight dimensions; see Section 7.2.2.) The colors show where different approaches have the lowest cost. It is difficult for a human developer to pick the "correct" combination of optimizations for a particular scenario, and, as we will show in Section 7.5.5, the cost of picking a suboptimal combination can be several orders of magnitude.

7.1.3 Our approach

Our proposed solution, Arboretum, is based on two key observations. The first is that having a large number of participants is both a challenge *and* a blessing: if each participant device helps just a little bit with the computation, this creates a massive pool of additional resources that can be used to process richer queries, even if they are beyond the reach of the aggregator alone. As previous systems in other domains, such as the NetSession CDN [306] or the PIER distributed query engine [158], have shown, this approach can create "organic scalability": adding more participants increases the resource demand, but also the resource supply. As with the earlier systems, the key question is how to efficiently distribute the

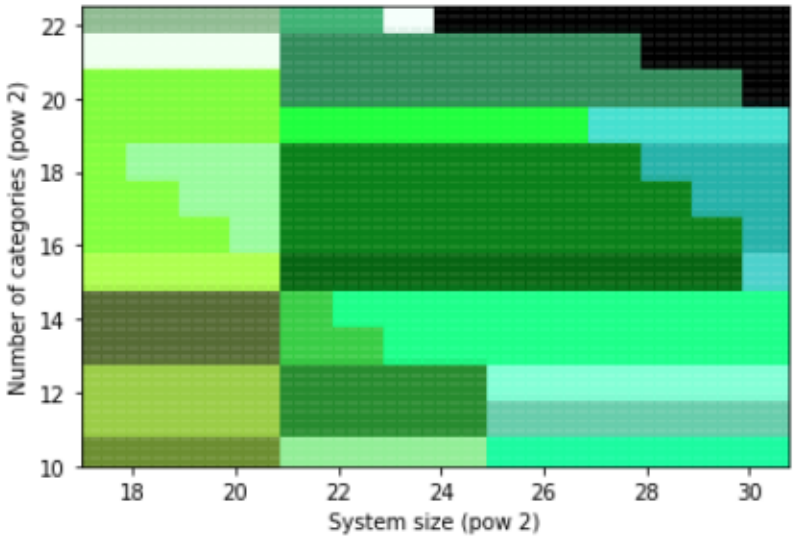


Figure 22: Part of the design space for the `top1` query. The colors show where different query plans have the lowest expected compute cost for participants.

work so that small devices, such as phones or laptops, can make a meaningful contribution; in Section 7.2, we show how Arboretum achieves this by breaking query plans into small, bite-size pieces that are each within the means of a small device.

The second key observation is that, although the design space is large and full of complex dependencies, it is also regular enough to be explored mechanically: high-level operators can be instantiated in different ways, the program can be segmented and transformed differently, and there are various parameters to be chosen. This is roughly analogous to query planning in a traditional DBMS, although of course the operators and transformations themselves are quite different. Thus, as we will show, it is possible to build a "query planner" for Federated Analytics that automatically finds a very good plan for most queries, without manual optimization or expert knowledge on the part of the analyst.

7.2. Query planning

We begin by describing how queries are formulated in Arboretum, and how Arboretum chooses a plan for each query.

Analysts write their queries in a simple imperative language (Section 7.2.1) that contains

```
stmt := stmt; stmt | var = exp | exp | var[exp] = exp —
        for var = exp to exp do stmt endfor |
        if expr then stmt else stmt
exp  := exp op exp | var | var[exp] | func(exp,...) | lit
op   := + | - | * | / | && | || | < | <= | > | >= | ! | ==
```

Figure 23: Arboretum's query language.

some high-level operations, such as `sum` or `max`. Each query is written as if it ran on a single machine that has access to the entire data set, without considering distributed execution or encryption. Along with each query, the analyst can specify limits on the costs she is willing to accept in a distributed solution, as well as an optimization goal (Section 7.2.2). Arboretum then verifies that the query is differentially private.

Next, Arboretum generates a (potentially large) number of query plans that can answer the query. It instantiates the high-level operations with various concrete implementations (Section 7.2.4), breaks the program into segments that can be executed by different parties (Section 7.2.5), and decides which kind(s) of encryption to use (Section 7.2.6). For each finished query plan, Arboretum estimates its costs (Section 7.2.7) and retains only the best plan that can operate within the limits the analyst has set.

7.2.1 Input language

Arboretum uses a simple imperative language for inputs that is loosely based on Fuzzi [302]. Figure 23 shows the syntax, which includes loops, conditionals, arrays, and the standard arithmetic and logical operators. The participants' input data is available as a predefined two-dimensional array: `db[i][j]` contains the j.th input from participant i. The program's output(s) are returned by calling the `output` function.

Arboretum supports several built-in functions: simple mathematical operations (`exp`, `clip`, etc.), aggregations over arrays (`sum`,`max`), a uniform sampling function (`sampleUniform`), the Laplace mechanism (`laplace`), and the exponential mechanism (`em`). As we will see in Section 7.2.4, some of these functions can be instantiated in several different ways. We use high-level operators in the input language because it helps with certifying differential

privacy, and also because we do not expect the typical analyst to know, or care about, low-level implementation details.

Figure 24 shows a simple example program (`top1`) that we will use as a running example. Each participant i belongs to one of several categories (say, hair color) and sets `db[i][k]` to 1 for the relevant category and to 0 otherwise. The program sums up `db` to obtain a vector of quality scores, which are simply the number of participants that belong to a given category, and then invokes the exponential mechanism to select a category, which it then returns. Notice that the program is written as if `db` existed on a single machine, and that no cryptography is being used. Distribution and encryption are handled transparently by Arboretum.

7.2.2 Constraints and goals

Along with the program, the analyst specifies an optimization goal and, optionally, a set of limits on the costs of solutions she is willing to accept. Our prototype supports six metrics that can be used to express these: two consider the aggregator (computation time and bytes sent), and the remaining four consider participant devices (expected and maximum computation time and bytes sent). For participant devices, the expected and maximum values differ because only a few devices are selected to serve on a committee, but these devices will have a higher cost. Other metrics, such as energy, should not be difficult to add if desired.

Arboretum discards any query plans that exceed the specified limits, and, among the remaining plans, returns the "best" one according to the chosen goal. For instance, an analyst could specify that the aggregator must not spend more than 1,000 core-hours and that user devices must not be asked to send more than 500 MB, and she could ask for the plan with the lowest expected computation time on participant devices.

```
aggr = sum(db);
result =
em(aggr);
output(result);
```

Figure 24: A simple example query (`top1`).

7.2.3 Verifying differential privacy

Once a query has been submitted, Arboretum's first step is to attempt to certify that the query is differentially private. Since this step is not the focus of our paper, we simply adopt a method from prior work – specifically the approach from Fuzzi [302], which is a good fit for our imperative query language and can certify many kinds of queries automatically, without help from the analyst. However, other approaches could be used instead; for instance, CertiPriv [32] would enable analysts to supply their own proofs of privacy, and thus allow Arboretum to accept queries where our automatic certification fails.

7.2.4 Program transformations

Once a query has been certified as differentially private, Arboretum transforms it into a query plan it can actually execute. This involves three steps: 1) replacing each abstract high-level operation, such as `sum` or `em`, with a concrete implementation; 2) deciding whether each step of the query should be performed by the aggregator, by a committee of devices, or by the participant devices themselves; and 3) adding suitable encryption to maintain confidentiality. Each of these steps can be done in several different ways, so, by trying all combinations, Arboretum can usually generate a large number of candidate plans for a given query.

We begin by discussing the first step. Many of Arboretum's abstract operations can be implemented in several different ways. One simple example is the `sum` operator that sums up the contents of an array: when the sum is computed by the aggregator, a simple for loop will do – but if the sum is computed by committees, a sum tree is better, because it can be spread across several committees. However, there is no single "best" degree! On the one hand, larger degrees will require fewer committees, so the cost of starting a committee

```
function em(s)
 L=max(s)-11; // for 16 bits
 for i=0 to len(s)-1 do
  if (s[i]>=L) then
   es[i]=exp((s[i]-L)*ϵ
   /(2*sens));
  else
   es[i]=0;
 r=random(sum(es)); s[0]=0;
 for i=0 to len(s)-1 do
  s[i+1]=s[i]+es[i];
  if ((r>=s[i])&&(r<s[i+1]))
   then return declassify(i)
```

```
function em(s)
 for i=0 to len(s)-1 do
  ns[i]=s[i]+
   Gumbel(2*sens/ϵ);
 x = 0;
 for i=1 to len(s)-1 do
  if (ns[i]>ns[x])
   then x = i;
 return declassify(x)
```

Figure 25: Two instantiations of the `em` operator, based on exponentiation (left) and on Gumbel noise (right), respectively.

can be amortized better and the *expected* cost is lower; on the other hand, larger degrees require each committee to do more work and thus lead to a higher *maximum* cost. A similar tradeoff exists with the `max` and `argmax` operators.

A more complicated tradeoff exists for the exponential mechanism. Figure 25 shows two possible instantiations of the `em` operator. On the left is a straightforward implementation of the textbook approach from [106, §3.4], which exponentiates the scores `s` to form an array `es`, then draws a random value `r` between 1 and `sum(es)`, and then returns the first category `i` such that the sum of the preceding elements of `es` is at most equal to `r`. Our only modification is that, since we have to work with finite-precision numbers, we normalize `es` to the range $[1, e^L]$ and ignore any elements with smaller scores; this results in (ϵ, δ)-differential privacy (see Appendix F.0.1). On the right is a variant, based on [99], that adds Gumbel noise to each score and then returns the element with the largest noised score. The tradeoff between these variants is complicated and depends on whether they are executed in FHE or using MPC. Notice that both variants invoke a `declassify` function to indicate that their result is safe to release in unencrypted form (see Section 7.2.6).

7.2.5 Basic type inference; Vignettes

Once all high-level operators have been instantiated, Arboretum performs type inference to assign to each variable and each expression 1) a basic type (`int`, `fix`, or `bool`), and 2)

a value range. The latter is important for deciding the parameters of the cryptosystems to be used (e.g., the plaintext modulus). The range bounds we infer are conservative; for instance, the lower and upper bounds for `a*b` are simply the products of the lower and upper bounds of `a` and `b`, respectively. However, the analyst can, if necessary, use the `clip` function to clip a variable to a smaller range.

Next, Arboretum decides which entity should execute each of the steps of the resulting program: the aggregator, a committee of participant devices, or a specific participant device. To this end, Arboretum breaks the program into short sequences of consecutive statements, which we call *vignettes*. The program thus becomes a sequence of vignettes, each of which is assigned to a particular entity. As a special case, a vignette that consists entirely of a data-parallel `for` loop can be *parallelized*, that is, its iterations can be assigned to different entities. For instance, a vignette that uses committees to compute a level of a `sum` tree can be parallelized, so that different committees compute the sum for different vertexes, and a vignette that encrypts the initial input data can be parallelized so that each participant device encrypts its own data.

As a first approximation, Arboretum tries all possible combinations of vignette boundaries and locations. This seems fine because we expect queries to be relatively short. However, we do implement a few simple heuristics to cut down the search space. We use branch-and-bound, by scoring the vignettes along the way (Section 7.2.7), and we discard partial solutions as soon as they exceed one of the analyst's limits or become worse than the best known solution. We do not allow constant assignments (such as `x = 0;`) to run in a vignette by themselves, and we do not allow consecutive vignettes to run in the same location, since they might as well be merged; the only exception is if both run on committees, which can make sense when there is a limit on the amount of computation a committee member may do.

7.2.6 Encryption-type inference

At this point, the program represents a distributed computation that returns the correct result; however, it does not yet ensure confidentiality, since the values are not yet encrypted. Arboretum's next step is to determine what needs to be encrypted, and how. This is done in three steps.

First, Arboretum identifies all values that need to be kept confidential. This includes anything that a) is derived directly or indirectly from the input database `db`, b) has not been passed through the `declassify` function, and c) is used in a vignette that runs on the aggregator or on individual participant devices. (Committees execute their vignettes using MPC, which already ensures confidentiality.) The only exception is that each participant device i is allowed to see its own input data `db[i]`. We use conservative taint tracking to find these values, starting from `db`.

Next, Arboretum adds encryption and decryption statements – initially without a specific cryptosystem. When a confidential value v is used in a participant or aggregator vignette, Arboretum inserts, right after the statement that creates v, a statement `v'=enc(v)` that creates an encrypted clone v'. It then replaces any instances of b in participant or aggregator vignettes with v'. When an encrypted value v' is passed to a committee vignette, Arboretum adds a statement `v''=dec(v')` at the beginning of that vignette and replaces any instances of v' with v''.

Third, Arboretum decides which cryptosystem to use for each value. If an encrypted value is only used in additions, it uses AHE, otherwise FHE. Whenever a cryptosystem is used for the first time, Arboretum inserts a key generation vignette at the beginning of the program and assigns it to a committee, to prevent any single entity from obtaining the private key.

Figure 26 shows, as an example, one of the candidates that are generated from the query in Figure 24 when there are 2^{30} participants and 10 possible outputs. Here, the `sum` operator has been instantiated with a simple AHE-based sum and the `em` operator has been

```
vignette (committee)
  ahePriv = aheKeygen();
  ahePub = pubkey(ahePriv);
parallel vignette (participant i)
  encdb[i] = aheEnc(db[i], ahePub);
vignette (aggregator)
  s = 0;
  for i=1 to 2^30 do
    s = s + encdb[i];
parallel vignette (committee i)
  ds[i] = aheDec(s, ahePriv)[i];
parallel vignette (committee i)
  ns[i] = ds[i]+Gumbel(2*sens/ε);
vignette (committee)
  x = 0;
  for i=1 to 10 do
    if (ns[i]>ns[x])
      then x = i;
  choice = declassify(x);
vignette (aggregator)
  output(choice);
```

Figure 26: One of the candidate programs that are generated from the query in Figure 24.

instantiated with a version that uses Gumbel noise. Notice that the Gumbel noise for each possible output is generated in a separate committee, and that a vignette has been added at the beginning to generate the AHE keypair.

7.2.7 Scoring

At this point, the candidate is complete. Arboretum now estimates the cost of running the candidate, to see whether it meets the analyst's constraints (Section 7.2.2), and to see whether it is better than the best known candidate so far. Scoring is based on a simple cost model, which we have built by benchmarking each building block – such as FHE operations, MPC start-up cost, incremental MPC costs for computations, etc. – on a reference platform. The model needs to be generated only once; after that, we can score a given query simply by adding up all the costs for the operations it performs.

This approach obviously does not yield the *exact* costs of running a query, for many reasons. For instance, the devices that are used by the actual participants could be different from our reference platform (or it could be a heterogeneous mix of devices) and the building blocks we use, such as the MPC frameworks, could apply their own internal optimizations

that could cause the total cost of a computation to differ from the costs of the individual operations. However, recall that we do not use scoring to predict the actual cost, but rather to weed out expensive candidates. Even a rough cost model should suffice for this purpose, although of course any inaccuracies could cause the chosen candidate to be somewhat more expensive than the 'true' optimal candidate.

7.2.8 Limitations

Like all query planners, Arboretum cannot necessarily find *the* best query plan – just the best plan it can generate using the primitives and optimizations it knows about. A human expert may be able to do better, e.g., by finding innovative optimizations, new algorithms, or specialized cryptographic techniques. However, Arboretum is fully automated, does not require expertise, and it can easily be extended with new optimizations if necessary.

Another important limitation comes from the fact that Arboretum's cost model can only be a rough approximation. In most cases, when query plan A outperforms query plan B on the reference platform that the model is based on, chances are that A also outperforms B on many other kinds of devices. But there may be exceptions, and in these cases Arboretum can potentially pick a suboptimal plan.

7.3. Query execution

In this section, we describe how Arboretum executes the query plan it has chosen.

7.3.1 Assumptions

For query execution, we adapted the approach from Honeycrisp (Chapter 4) and, as a result, Arboretum shares all of Honeycrisp's nine assumptions: 1) a locally generated keypair σ_i, π_i on each device i, 2) a rough lower and upper bound on the number of participating devices, 3) an immutable bulletin board for broadcasting a small amount of data, 4) an external, time-stamped channel for reporting misbehavior, 5) secure, authenticated point-to-point channels, 6) an efficient hash function that is indistinguishable from a random oracle, 7) an upper bound f ($\approx 1 - 5\%$) on the fraction of malicious devices, 8) an upper bound g on the probability that a device goes offline while participating, and 9) an one-off randomness

beacon that generates a small number of truly random bits B_0 when the system is first launched.

7.3.2 Sortition

When query execution begins, Arboretum must first choose the devices that will serve on each committee. We generalize the sortition mechanism from Honeycrisp, which is in turn based on an idea from Algorand [128]. A key difference to Honeycrisp, which has only one committee, is that the committee size m depends on the number of committees c. To guarantee privacy, we need an honest majority in *all* c committees, which happens with probability at least $p := 1 - 2c \cdot e^{-fm}(2ef)^{\lfloor m/2 \rfloor}$. Since the number of committees can vary between query plans, Arboretum calculates the minimum committee size for a given query plan before scoring it, so it can estimate the cost of the MPCs correctly.

7.3.3 Phases and message passing

Once sortition is complete, the computation proceeds in rounds. Each round executes one particular vignette, but parallel vignettes (Section 7.2.5) can be executed concurrently by several devices or committees. We use a *verifiable secret redistribution* (VSR) scheme [136], like in Mycelium (Chapter 6), to securely transfer keys from one committee to the next. This scheme generates and distributes newly random secret shares from the existing ones, such that shares from *different committees* cannot collude to recover cryptographic keys, and malicious members cannot obstruct honest committees.

When vignettes are completed, the aggregator serves as a "mailbox" by accepting these outputs and making them available to the next vignette. Outputs are signed with the sender's key and encrypted with the recipient's key, so the aggregator cannot view these messages' contents. It can corrupt or drop them, but it could only harm itself by doing this (by causing the query to abort and not return any results, which we assume the aggregator is interested in); it would not affect privacy.

7.3.4 Verifying participant vignettes

Vignettes that are executed by committees naturally maintain integrity through the MPC protocol. However, if a vignette is assigned to individual participant devices, some of these devices may execute it incorrectly. To guard against this, just like in Honeycrisp, Arboretum requires participant devices to submit a zero-knowledge proof along with the results of any vignette they have executed directly (that is, not via MPC). The aggregator checks these proofs and ignores any inputs from devices that fail to provide a correct proof.

7.3.5 Verifying the aggregator's steps

Another possibility is that the aggregator itself could be malicious and could alter the steps of any vignettes that have been assigned to it. Since the aggregator does not hold any private keys, it cannot directly learn any confidential information by doing this; however, there are ways to do so indirectly. For instance, if the aggregator was supposed to add up AHE-encrypted inputs from 1,000 participants and pass the sum to a committee for Laplace noising and decryption, a malicious aggregator could discard all inputs except one, and add that input to itself 1,000 times. Since in this case the noise would be calibrated to the original sum, the aggregator has effectively shrunk the noise by a factor of 1,000.

To guard against this, Arboretum requires the aggregator to, after executing a vignette, just as in Honeycrisp, build a Merkle hash tree (MHT) with the results of the individual steps in the leaves (excluding only the final `output` step). Each participant device then picks some leaves at random and challenges the aggregator to return a) the contents of this leaf, and b) an inclusion proof (i.e., a path from that leaf to the root of the MHT). This is safe because the aggregator operates on encrypted data, just like individual participants. The devices then verify the steps they have audited. The number of leaves each device audits is chosen such that the probability of missing an incorrect step is smaller than some threshold p_{max}, which is a system parameter and is chosen when the system is first launched. For a security analysis of Arboretum, please see Appendix F.3.

7.4. Implementation

For our experiments, we implemented a prototype of Arboretum's query planner in C++; this prototype has 11,787 lines of code. We provide some key details below.

Cryptosystems: Our prototype uses the BGV cryptosystem [57] for FHE. The specific parameters depend on the encryption types the query planner infers (Section 7.2.6), but a typical query with one-hot encoding uses a plaintext modulus to 2^{30} (enough to sum binary values across one billion users), a 135-bit prime for the ciphertext modulus, and a polynomial degree of 2^{15}. This results in over 256 bits of security [14].

MPCs and ZKPs: For multi-party computation, we use the MP-SPDZ framework [168] – specifically, the SPDZ-wise Shamir program, where operations are performed in a finite field with a configurable prime modulus. This is a natural fit for BGV key generation and decryption, and it provides security against malicious parties as long as there is an honest majority. For efficiency, the encryption, decryption, and key generation MPCs set the prime modulus to BGV's ciphertext modulus. For the zero-knowledge proofs, we use the ZoKrates toolbox [1], with `bellman` [2] as the proof system.

Secrecy of the sample: We implement secrecy of the sample (Chapter 2.1) as follows. Let $\frac{x}{b}$ be a fractional approximation of the sample size, where b is the total number of bins in a standard ciphertext – for instance, setting $x = b/2$ would sample 50% of the participants. First, a committee samples a value j uniformly at random from 1 to b. Each participant device also randomly chooses an index i from 1 to b, and places their encrypted local input only in that i-th bin, setting all other bins to 0. They upload the result to the aggregator as usual. To sample from the desired range, the committee only decrypts bins from j to $j+x-1$ (modulo b) – they can do this by summing over all bins in the aggregate ciphertext, but replacing the bin values with 0 when they fall outside this range. We discuss security of this protocol in Appendix F.2, but in essence, participant devices do not know which

[1] https://github.com/Zokrates/ZoKrates
[2] https://github.com/zkcrypto/bellman

random value has been sampled by the committee (so they can't force themselves to be included or excluded, or even know if they were sampled), and neither the committee nor the aggregator knows which bins the participant devices selected, so secrecy of the sample is preserved.

Precision: In MP-SPDZ, we use the `cfix` and `sfix` fixpoint types for operations with non-integer values. We set the fixpoint length to be 30 bits for the integer part and 16 bits of precision for the decimal part, which gives 40 bits of statistical security in the MPC programs. The use of fixpoint types avoids some of the complications with implementing differential privacy, such as irregularities due to floating-point implementations [216]; we additionally use base-2 for the exponential mechanism, as suggested by Ilvento [160], which also has better support in MP-SPDZ. As with most other implementations, the use of finite-range data types adds a small δ to the guarantee, since the tails of the Laplace and Gumbel distributions are cut to the representable value range.

Cost model: Our prototype includes a cost model that is based on benchmarks of the various cryptographic primitives on a Dell PowerEdge R430 server with two 2.4 GHz E5-2620 CPUs and 64 GB of RAM. This model cannot exactly predict the costs of a query on a heterogeneous mix of devices, but it should be sufficient for ordering the solutions (see also Section 7.2.8). For primitives with many settings, we benchmarked some representative settings and interpolated the others. Our model does account for some simple MPC optimizations, such as the fact that the first comparison is more expensive than subsequent comparisons because it requires the generation of multiplication triples. We include validation data for our model in Appendix F.4.

7.5. Evaluation

In this section, we report results from our experimental evaluation of Arboretum.

7.5.1 Supported queries

Arboretum extends the range of queries that can be answered at scale, but it also generalizes some existing solutions. To show this, we have chosen a mix of new and old queries for our

Query	Action	From	Lines
`top1`	Most frequent item	[106]	3
`topK`	Top-K selection	[99]	8
`gap`	Exp. mechanism with gap	[95]	8
`auction`	Unbounded auction	[207]	7
`hypotest`	Hypothesis testing	[65]	12
`secrecy`	Secrecy of sample	[28]	16
`median`	Median	[51]	39
`cms`	Count-mean sketch	[263]	5
`bayes`	Naïve Bayes	[264]	16
`k-medians`	K-Medians	[264]	30

Table 3: Supported queries.

evaluation, which is shown in Table 3. The first six queries are new: the first five use the exponential mechanism, and we are not aware of any other system that can answer them efficiently with billions of users and without a trusted aggregator. The remaining queries are adapted from earlier work: `cms` is the query from Honeycrisp, `bayes` and `k-medians` are two queries from Orchard, and `median` is the query from Boehler et al. [51], which can be easily extended to support quantiles. Since Orchard's query language is functional, we rewrote these queries in Arboretum's language; the other two systems are for specific queries and do not have a query language, so we implemented their queries in Arboretum. Table 3 also shows the number of lines for each query, to show that they can be formulated concisely in our language. For categorical queries (the first five in Table 3), we use a one-hot encoding for the categories. Our implementation of the `median` query also uses one-hot encoding and differs from the one in [51] in a few other details; we provide more information in Appendix F.1.

7.5.2 Experimental setup

Since we cannot actually create a deployment with billions of devices, we follow the approach from all prior chapters, and extrapolate the costs based on benchmarks of the individual operations. For instance, the zero-knowledge proofs the participants submit at a particular stage of a query are structurally identical and can be verified independently, so we measure the time t it takes to verifying one such proof and then estimate that it would take an aggregator with γ cores $\frac{N \cdot t}{\gamma}$ to verify N of them. Whenever possible, we use direct

measurements. For instance, the MPCs involve around 40 participants, so we benchmark them on a cluster of five PowerEdge R430 servers, each with twelve cores (2xE5-2620 at 2.4 GHz), 64 GB of RAM, and running Fedora Server 34; to reduce interference, we pin each participant process to a separate core.

To simplify the comparison with Honeycrisp and Orchard, we use the same key parameters, unless stated otherwise: $N = 10^9$ participants, up to $f = 3\%$ malicious participants, and a $2 \cdot 10^{-9}$ probability of privacy failure after running 1,000 queries. This leads to committee sizes of about 40 members (depending on the number of committees), which is the setting Orchard uses. We use C=1 categories for the `hypotest` and `cms` queries, C=10 for the `kmedian` query, C=115 for the `bayes` query (as in Chapter 5), and C=2^{15} for the other queries. For `topK`, we used $k = 5$ to return the five most common items. To achieve a fair comparison to Orchard and Honeycrisp, which use SCALE-MAMBA for MPC, we reimplemented the MPCs for `cms`, `bayes`, and `k-medians` in MP-SPDZ.

7.5.3 Cost of running queries

We begin by examining the cost of running the queries in Table 3. We allow participant devices to send up to 4 GB of traffic and spend up to 20 minutes of computation time, and we limit the aggregator's computation to 1,000 core hours.

Figures 27(a) and 27(b) show the *expected* bandwidth and computation required of each participant, respectively; there is one bar for each query, which includes both the cost for normal participant-side computations and the expected cost of serving on a committee (that is, the actual cost of each committee type multiplied by the probability of being selected for a committee of that type). As expected, the arboretum/figures show that the exponential mechanism has a much higher cost than the Laplace mechanism; the cost is particularly high for the `topK` query, which has to find the highest quality score k times. Nevertheless, the expected costs are low in absolute terms: each participant sends between 132 kB and 3 MB of data and spends between 7.1 s and 62.4 s of computation time.

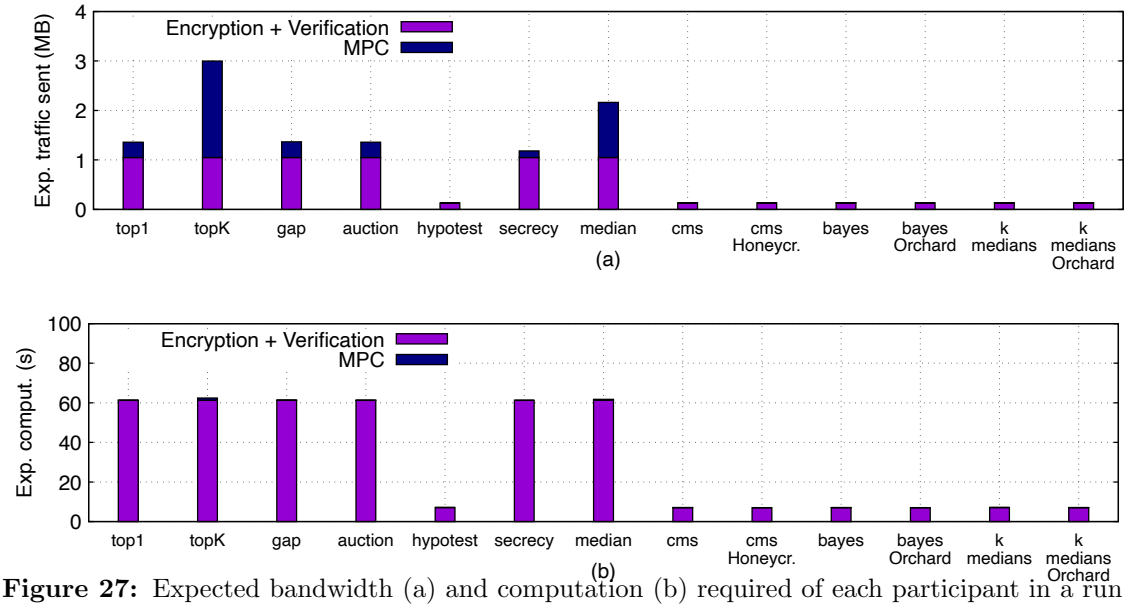


Figure 27: Expected bandwidth (a) and computation (b) required of each participant in a run of each algorithm.

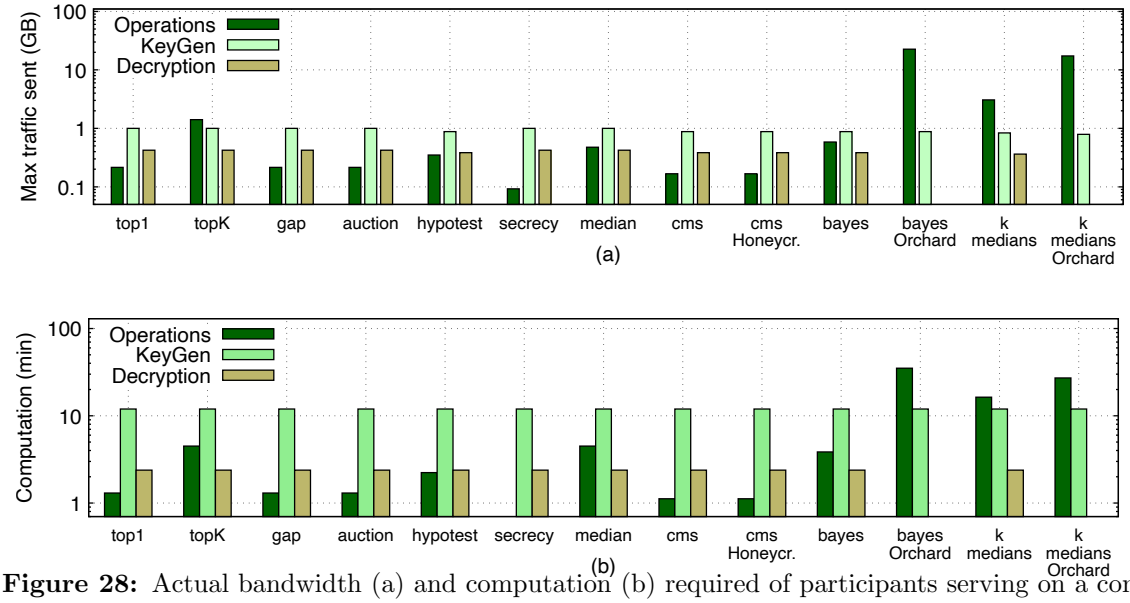


Figure 28: Actual bandwidth (a) and computation (b) required of participants serving on a committee, by committee type. The percentage of total participants which serve on a committee of any type ranges from 0.00022% (`k-medians`) to 0.49% (`topK`).

Of course, if a participant is actually selected to serve on a committee, its actual costs are much higher than the expected costs; the precise amount depends both on the query and on the committee type. Figures 28(a) and 28(b) show these costs, with separate bars for each committee type. For most queries, the key-generation committee is the most expensive; it consumes roughly 1 GB of traffic and 12 minutes of computation time. The two exceptions

are the operations committees of `topK` and `k-medians`, which consume 1.4 GB and 3 GB of traffic, respectively. Although these costs are higher, they are still well within the means of a typical device (especially if the computation is done at night, while plugged in), and the odds of being selected for a committee are very low: for instance, with 10^9 participants, the `topK` query has one 42-member committee for key generation, 328 for decryption, and 115,334 for operations such as noising and computing the argmax. Thus, in a given run of `topK`, only 0.49% of the participants are serving on a committee of any type.

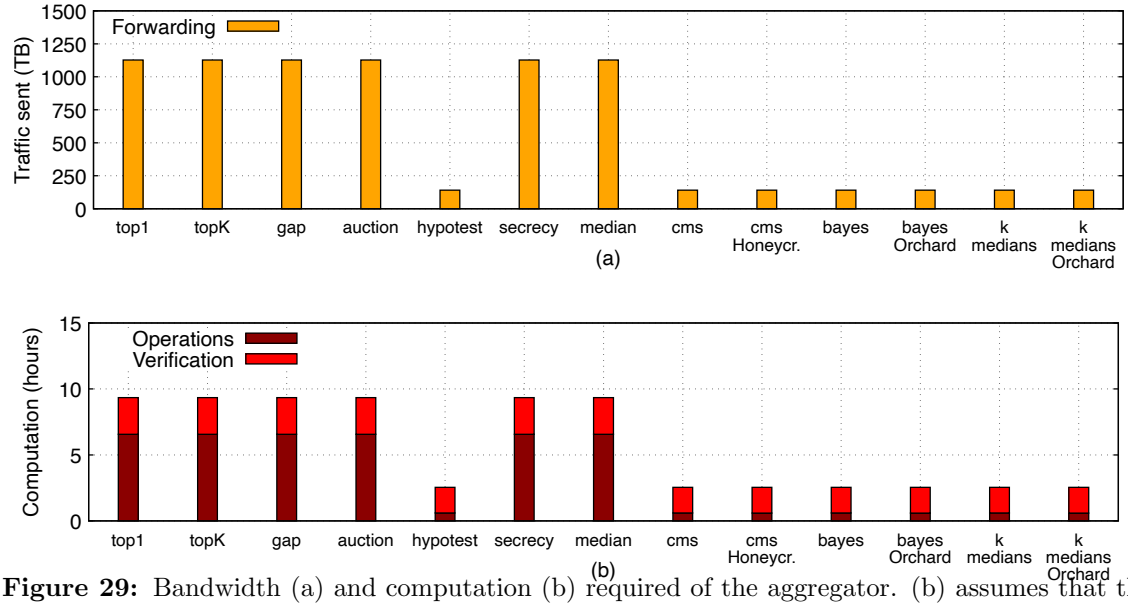


Figure 29: Bandwidth (a) and computation (b) required of the aggregator. (b) assumes that the aggregator has 1,000 cores.

Figures 29(a) and 29(b) show the cost for the aggregator. Once again there is a clear difference between the exponential and Laplace mechanisms: the former involves more committees, so more bandwidth is spent on forwarding. (`hypotest` is an exception here because it has only a single category.) The bandwidth costs are high in absolute terms, but, on average, each of the 1 billion participants just receives about 1.1 MB, which is the size of a small image file. The computation time is below 10 hours when 1,000 cores (about 10 powerful servers) are used; most of the tasks are trivially parallelizable, so the time could be reduced by using more cores.

In the case of queries we took from Honeycrisp and Orchard, the arboretum/figures also

show the costs of the original system. These costs are almost identical to Arboretum's in expectation, however, the cost for committee members was much higher (∼20 GB of traffic and 35 minutes of computation time) since Orchard does not leverage multiple committees. Notice that the original systems were custom-designed for these queries, whereas Arboretum was able to find these query plans independently, without human intervention.

7.5.4 Cost of query planning

Before Arboretum can run a query, it first needs to choose a good plan. As discussed in Section 7.2, this involves generating and scoring many different candidate plans. To see how expensive this step is, we ran the query planner on a PowerEdge R430 server for each of the queries from Table 3, and we measured the runtime until a plan was chosen.

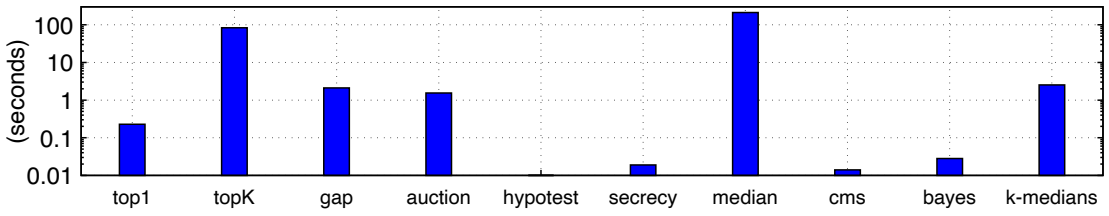


Figure 30: Runtime of the query planner.

Figure 30 shows our results. The runtime varies widely, from 10 ms (`hypotest`) to 212 s (`median`); this is because the more complex queries have a larger design space, with more possible combinations of operator expansions, more ways to divide them into vignettes, and more combinations of placement decisions. For instance, in the case of `median`, the query planner considers 1,251,001 different plan prefixes and 16 full candidate plans before it makes a decision. Since the queries themselves take hours to run (Section 7.5.3), it seems fine to spend a few minutes on planning.

To test whether our branch-and-bound heuristics are effective, we also ran the query planner with these heuristics disabled. This caused the planner to run out of memory for half of the queries; in the cases where it did terminate, it took between one and three orders of magnitude more time.

7.5.5 Benefits of query planning

Next, we examine whether the above costs are "worth it" – that is, whether query planning really does yield nontrivial benefits. This question is hard to answer without looking at the *entire* design space, which is enormous, but we can at least provide some illustrative examples. To that end, we ran the query planner on the `top1` query, using various combinations of the system size N and the number of categories C. For each combination, we recorded a) the query plan that was chosen, and b) the costs of running that plan.

The choices the query planner made are shown in Figure 22, which we had used as an illustration in Section 7.1.2. As discussed in that section, there are lots of different design choices, and the optimal solution for different regions in this space requires a subtly different combination of them.

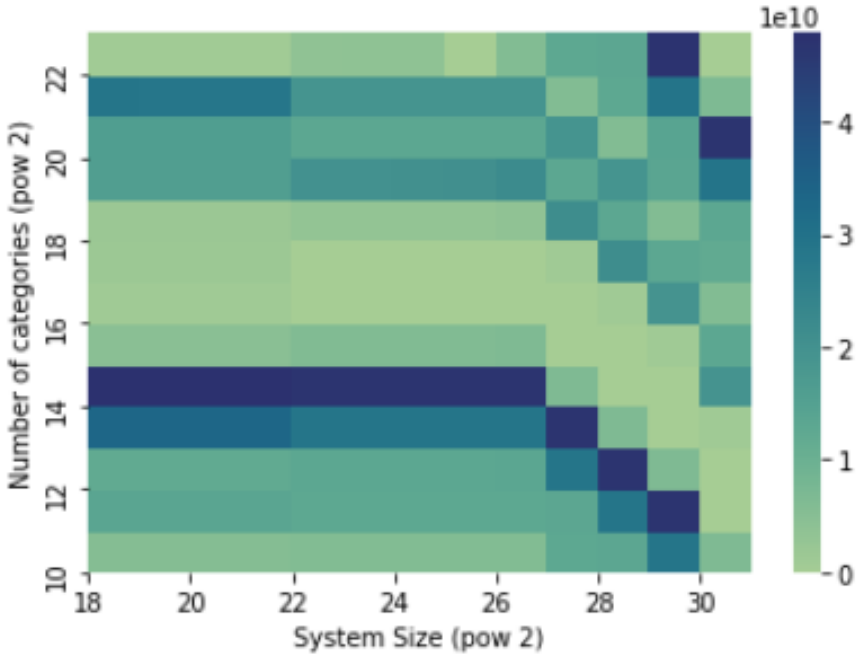


Figure 31: Additional traffic sent by any participant in the worst case, using an average query plan instead of Arboretum's optimized choice from Figure 22.

Even if the query planner makes lots of different choices, this does not necessarily mean that it is saving a lot of overhead – if the differences are small, a single fixed choice could perform almost as well. To quantify the actual benefit of the query planner, we generated 100 random

plans for `top1` and estimated the costs of running these plans for every combination of the above two parameters; we then subtracted the costs of the best query from Figure 22 from the average costs amongst the random plans, and plotted the results as a heatmap in Figure 31. The cost differences vary considerably, but in general, a single fixed plan would often require the participants to send gigabytes of additional traffic, relative to Arboretum's optimized choice.

7.5.6 Scalability

A final question is how well Arboretum scales to large numbers of participants, and whether the scaling behavior is qualitatively different from that of earlier systems. To examine this, we used the `top1` query as an example and generated query plans for a wide range of system sizes, from $N = 2^{17}$ to $N = 2^{30}$ participants.

Figures 32(a-c) show our results: (a) shows the aggregator's computation time, and (b) and (c) show the average and maximum computation time for participants, respectively. Each graph contains three lines: two with different limits on the aggregator's computation time, and one without any limit. The overall pattern is similar to Orchard: the cost for the aggregator increases with N because it checks all the ZKPs and, at least initially, also sums up the participants' contributions; the participants' expected cost decreases with N because the chance of serving on a committee decreases. However, if we add limits, the picture changes: at some point, the aggregator has to outsource some of the computation to the participants, whose expected cost increases accordingly. (With the lowest limit, the aggregator cannot even afford to check ZKPs after $N = 2^{28}$ anymore, so the red line stops there.) This option would not be available in the earlier systems, which use a single committee.

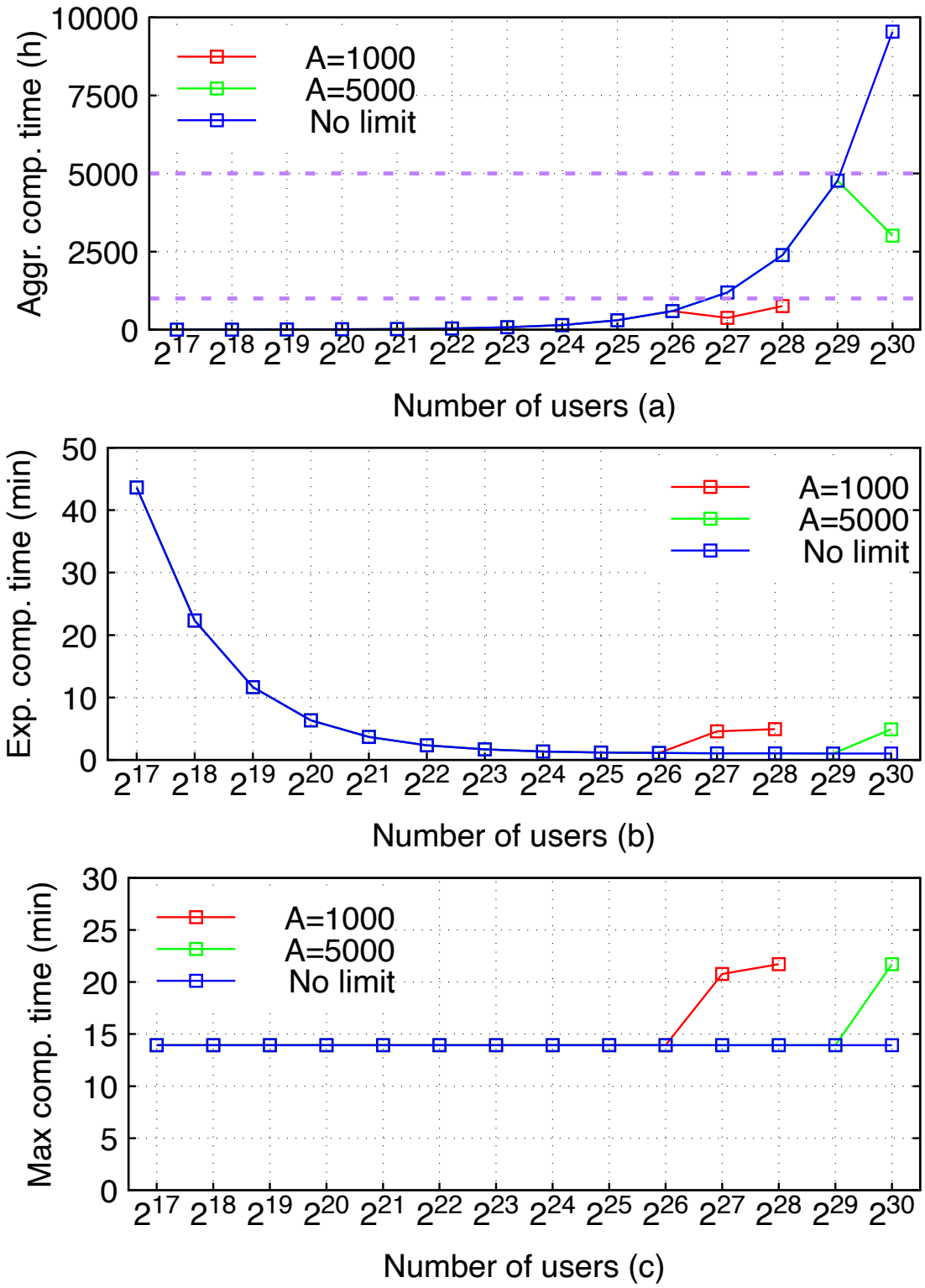


Figure 32: Scalability of computation cost for the aggregator (a), and expected (b) and maximum (c) cost for participants,varying aggregator compute cap A.

7.6. Related Work

Related work on general private analytics at scale has already been discussed in Chapter 3; here we discuss prior work that is specifically relevant to Arboretum.

Secrecy of the Sample: Sampling from a large set of clients in order to boost privacy is used in several differentially private algorithms, most notably differentially private SGD [3]. However, previous implementations of this sampling [10, 242] give guarantees in the local model. We are not aware of any prior solutions for large-scale federated settings.

Single-committee systems: The three previous chapters in this thesis, Honeycrisp, Orchard, and Mycelium, all share Arboretum's approach of outsourcing certain computations to MPC committees. However, all three systems are restricted to a *single* committee that performs just a few simple steps (key generation, noising, and decryption). When running the exponential mechanism with more than a trivial number of possible outputs, this committee quickly becomes a bottleneck. A fourth system, Böhler and Kerschbaum [51], also uses a single committee to implement the exponential mechanism but targets smaller deployments; it was shown to scale up to one million participants.

Query planning: Arboretum is not the first system to use query planning to speed up privacy-preserving analytics. Conclave [289] and SMCQL [33] generate query plans that combine local cleartext processing with small MPC steps, but they are both designed for a scenario with a *small* number of participants that each hold a *large* amount of data, which is the exact opposite of what we focus on here. For example, Conclave's MPC steps are limited to at most three parties, and SMCQL supports only two. Opaque [308] assumes that the private data is spread across multiple servers, but that these servers belong to a single organization and are part of a single cluster; Arboretum, in contrast, assumes a federated setting in which each device is owned by a different person.

7.7. Conclusion

Arboretum significantly expands the range of queries that can be answered efficiently using Federated Analytics at massive scales: it adds support for the exponential mechanism and

for secrecy of the sample. To the best of our knowledge, Arboretum is the first practical system that can handle billions of users and thousands of categories in the federated model. Since such queries require an enormous amount of computation – in some cases more than even a very powerful aggregator could handle – Arboretum provides a way to leverage small contributions from the massive participant base to generate enough resources, with the aggregator acting as a central orchestrator.

While the increasing number of options for Federated Analytics is mostly good news, it does increase the complexity for the analyst, who has to make the correct choices for each particular query to get good efficiency. Arboretum solves this problem by separating the high-level specification of the query from the detailed query plan, which can be chosen automatically – roughly analogous to compiler optimizations or to DBMS-style query planning. Our results show that the resulting plans are competitive with the hand-optimized plans from prior work; thus, the analyst can safely leave most choices to the planner.

CHAPTER 8 : Conclusion and Future Work

This thesis broadly attacks the problem of secure aggregation in large-scale distributed data analysis: how to aggregate distributed sensitive data to a central location without viewing the underlying data outside of its aggregate form. This problem is already practical - companies like Apple, Google, and Microsoft all have deployed systems to collect private statistics from their user bases. Despite this, there are yet to be standard solutions. This work is situated among very few other secure aggregation techniques, including ones from Google [35, 54] and Mozilla's implementation of Prio [79]. These designs all have different tradeoffs with respect to efficiency, robustness, and scalability.

The emphasis in this thesis lies beyond secure aggregation and towards support for differential privacy. In this vein, the work in this thesis is unique because of how it uniquely tailors system design towards differentially private analytics. This is a break from previous work on secure aggregation, which is focused purely on aggregation, and often tacks on differential privacy as an optional addition after data has been aggregated (e.g., in Prio [79]). Incorporating differential privacy as a prerequisite changes the architectures of the systems we design, in particular for Orchard and Arboretum, which show how to support a wide variety of DP algorithms and mechanisms.

This direction of work is a critical component for next-generation private analytics systems. As differential privacy gains steam as the most widely accepted technical definition of privacy, there need to be solutions to not only analyze and release private data, but keep data secure and private throughout the collection, storage, and analysis processes. This thesis is one first step in this direction, showing that this is possible at massive scales, which has been a particular challenge, but a necessary one, given the huge scale of systems that collect our data today. Cryptographic primitives, when used on their own, don't scale to this level of billions of users, partly because of the very pessimistic threat models they traditionally assume (up to 1/3 or 1/2 of devices may be faulty!) The OB+MC threat model we intro-

duce in this work shows that a more realistic assumption can make more progress possible, and support a reasonable range of queries. However, the span of queries we can support is still fairly small when considering the space of all possible analytics techniques that can be deployed at scale. A large area of future work could therefore, first and foremost, expand the sorts of algorithms and datasets that can be processed at scale by introducing additional configurations and scalable cryptographic building blocks. While Arboretum, for instance, shows how to handle large categorical data, very large (and particularly, sparse) vectors of data remain a challenge to process at scale. There are still many other kinds of data that our systems do not support, including most non-relational data (outside of some Pregel-style queries supported by Mycelium). As the differential privacy community continues to develop more sophisticated DP techniques, systems advances will be necessary in tandem to allow support for their deployment at scale. Future questions in this space include whether we have to carve out more classes of queries one by one, with custom solutions, or whether there are more general techniques that can handle broader classes of algorithms, as we begin to introduce in this thesis.

Moreover, the Honeycrisp-style system design of offloading computation to committees introduces more overhead than might be desirable, particularly from those devices designated as committee members. While their costs are orders of magnitude lower, there is not an insifignificant cost for regular participants as well, who must pay extra for tasks such as verification. In practice, our systems' reliance on multiple rounds of iteration could also prove challenging, as high rates of user dropout over time would reduce overall accuracy and effectiveness of these systems. Future work could take advantage of opportunities to increase efficiency, for instance, more clever or aggressive sampling of participants to reduce the overall data load, optimized verification protocols for particular kinds of queries, or selecting committee members more dynamically based on their computational resources and device properties. Each of these options could drastically lower participant overhead and make the systems described in this thesis easier to deploy in practice.

There are also, at this point in time, many different variations of the standard differential privacy definition (see [90]), and other neighboring definitions that we don't touch on in this thesis (e.g., user-level DP as opposed to row-level DP). Incorporating some of these definitions could simply require different amounts or shapes of noise, or more sophisticated privacy accounting, which all could be relatively simple changes to committee computation. Others would involve more fundamental challenges – implementing user-level DP, for instance, could require correlating contributions from the same individuals over time while still somehow maintaining anonymity, as opposed to how we currently treat all contributions in identical ways as completely anonymous. This is further complicated by notions of group privacy, correlated data, and privacy dependencies [30], for which differential privacy does not necessarily apply (at least not naïvely). An opportunity for future work would be to tailor system design to these different variants, or offer support for privacy definitions that better capture the interrelated nature of our informational world, including perhaps, approaches to privacy that are less technical (see, for instance, contextual integrity [229].)

APPENDIX A : Parameter Choices

The detailed explanation of the parameters in Section 4.6.2 is as follows: for each protocol, to achieve the same error α for utility with probability at least β, we must set the following values for privacy parameter ϵ, where k is the number of total queries over the course of the systems' lifetime and c is the amount of times the count is significantly updated during this period. In this case as mentioned above, we set $\beta = 0.95$ to allow for 5% error rate, $c = 40$ which corresponds to a change every 91 days (3 months) over a 10-year period, and $\alpha \sim 10^{-6} \cdot N = 5000$ which corresponds to the range that the query can take assuming a 1% bound on error (however, we note that this error bound is a linear component that affects all systems equally - so if this error bound changes, all systems will suffer equally in privacy budget). This also requires an assumption on the results of the query, which we set to 0.001% of the population given the humongous user base setting we are operating in. For GDP and Honeycrisp, we set k assuming a vector of at most 10 unique 1-sensitive data points collected from each individual that responds to the query.

The explicit formulas used for calculating privacy budget over time are as follows:

$$\text{LDP: } \epsilon = O(\frac{\sqrt{N \ln(1/\beta)}}{\alpha} \cdot k) \text{ [67]}$$

$$\text{GDP: } \epsilon = O(\frac{\ln(1/\beta)}{\alpha} \cdot k) \text{ [106]}$$

$$\text{Honeycrisp: } \epsilon = \frac{8c(\ln k + \ln(2c/\beta))}{\alpha} \text{ [106]}$$

The bound for Honeycrisp could be further improved by [199], which discusses implementing SVT in practice.

APPENDIX B : Options for Generating Randomness in Honeycrisp

Many MPC protocols can be made significantly more efficient if the users share correlated randomness (e.g. [34, 82, 169, 170]). This correlated randomness can be distributed (by a trusted dealer) or generated by the users themselves in a "pre-processing" phase that is independent of the users' inputs. In this pre-processing model, almost all of the communication cost of the protocol can be moved to the pre-processing phase, and the "online" phase of the protocol can be made extremely efficient.

Thus, optimizing the MPC portion of our protocol requires optimizing the pre-processing phase of the protocol. By default, in Honeycrisp the committee members generate the randomness themselves, as in [169, 170, 176]. This approach is the most secure, since no trusted party is needed at all; however, it is also computationally expensive and requires significant amounts of bandwidth and RAM (see Section 4.6.4), so it will probably not be feasible on cell phones today. It seems realistic on laptop computers, however, so Honeycrisp could simply restrict committee membership to sufficiently capable devices. Note that the committee is very small, relative to the entire device population, so most devices will never serve on a committee at all, and hardly any will serve more than once.

If a party with some very limited trust happens to be available, Honeycrisp can benefit from it and reduce the overhead for the committee. We briefly sketch two options for this: 1) A trusted execution environment (e.g., Intel SGX) could be used to generate correlated randomness and distribute it to the participants. This is extremely efficient from a computational standpoint, but it introduces Intel as a trusted source, and is potentially vulnerable to side-channel attacks [59, 139, 218]. 2) A semi-trusted dealer could generate and distribute the randomness. The entity would never see any actual user data or participate in the MPC in any way, would just need to run one computation to generate random bits, and send these random bits securely to the parties participating in the MPC. For our evaluation, we chose the default option, without any trusted parties.

APPENDIX C : Honeycrisp Security

In this section, we demonstrate that Honeycrisp satisfies the following properties:

1. **Privacy.** The system remains ϵ-differentially private for a given ϵ, or else everyone learns, with high probability, that the Aggregator cheated.
2. **Correctness.** When the Aggregator receives a response to a query, that response is correct – that is, the exact answer plus the noise required for ϵ-differential privacy.
3. **Liveness.** As long as there is sufficient privacy budget left, the Aggregator will continue to be able to query the system and receive responses with high probability.
4. **Indemnification.** If the Aggregator follows the protocol, devices cannot fabricate evidence that would prove that the Aggregator had deviated from the protocol.

For the Privacy guarantee, we assume that the Aggregator may act maliciously, including by (statically) corrupting up to proportion f of participating devices. However, for the Correctness, Liveness and Indemnification protocols we specifically assume that the Aggregator follows the protocol. We believe this is a reasonable assumption since the primary purpose of the protocol is to provide data to the aggregator, thus the aggregator has no incentive to undermine liveness.

Since the first property and the latter three depend on entirely different assumptions about who may behave maliciously, we will keep these as separate as possible in the proofs below.

C.1. Assumptions

We list our assumptions below and for each provide some intuition for how the assumption could be realized or how it is already realized.

Assumption 1. Each device i has a locally generated keypair σ_i/π_i for signing messages; the aggregator can check whether each public key π_i belongs to a valid device.

This could be implemented by using public keys stored in secure enclaves. In Apple's case, many Apple devices already use Apple-designed chips which support secure enclaves. One

way to allow asserting that a public key belongs to a valid device is for each enclave to contain a signature of its public key under Apple's public key.

Assumption 2. There is a once-off randomness beacon – an independent party P that can be trusted to generate a single random string, B_0, when the system is first launched.

This could be implemented by having a widely respected entity provide the random number, or by depending on a trusted physical randomness source, such as a state-sponsored lottery. If P is an entity that provides public randomness already [1], then P does not need to do any additional work to be used by Honeycrisp. This only has to be used *once* to choose the first block, B_0.

Assumption 3. All devices know an upper bound N_{max} and a lower bound N_{min} of the number of potential participating devices in the system.

If the true number of devices is N_{tot}, then by definition:

$$N_{min} \leq N \leq N_{tot} \leq N_{max}$$

We assume $\frac{N_{max}-N_{tot}}{N_{min}}$ is always below some constant (low) threshold (this determines the portion of Sybils $\mathcal{A}$ could make without getting caught). $\frac{N_{max}}{N_{min}}$ should also be below some (more generous) constant threshold.

In Apple's case, Apple provides estimates on the number of devices sold, with current figures estimating upwards of a billion total devices [219], as well as estimates on the current installed base [296] of 900 million. This provides the upper bound. Subtracting an estimate of the maximum number of such devices that could have gone offline permanently or may not be online in a given round provides the lower bound.

Assumption 4. There is an immutable bulletin board, B, that the aggregator can use to broadcast a small amount of data to all devices.

[1] e.g., https://www.random.org/

The aggregator could simultaneously publish to several (free, centralized) "bulletin boards", e.g., Wikipedia, StackExchange, Reddit. Alternatively, this could be implemented by any external distributed ledger. Since it is only used by the aggregator, it is acceptable if posting to the ledger incurs a small fee. For example, storing a 256-bit string on the Ethereum blockchain costs 20000 gas [297]. At current prices, that translates to roughly $.03 USD.

Assumption 5. Devices can use an external, time-stamped channel, X, to report the aggregator if it behaves maliciously.

This could again be satisfied by any of the (free, centralized) "bulletin boards" noted in the previous assumption. However, this could also be implemented by an external entity such as a newspaper with a dedicated editor. Once a device writes to this channel, the newspaper would be able to reach out to the aggregator, and would require a reasonable response within a time frame. If a malicious aggregator has been confirmed, the newspaper can publicly notify all users. This would not require a large volume of messages, since only one is enough to trigger an action.

Assumption 6. Secure, authenticated, point-to-point channels can be established from each device to a) the aggregator, and b) a small number of other devices.

Secure point-to-point channels with very high probabilities of low latency are now common with secure connections on the internet. If a device does not have a good connection that satisfies this, they can be considered as offline. It is assumed that Apple has sufficient resources to stay online.

Secure channels between devices are only needed within the committee. These can be achieved with TLS channels. If devices are behind a NAT firewall, an external VPN service could be employed to allow communication.

Assumption 7. There is an upper bound f ($\approx$ 1–5%) on the fraction of participating devices that may be malicious, collude with each other, or collude with the aggregator.

Apple has methods to determine whether Apple software is running on an Apple device. Therefore, non-aggregator adversaries would only be able to gain identities in the system by actually buying physical Apple devices. Buying out any significant portion of the number of devices would be very expensive. Furthermore, in the case of Apple, there are extensive measures used to control the distribution of software. As such it is challenging to run malicious (non-approved) code on devices or to access non-user facing data. The vast majority of users will not make the efforts to overcome these challenges. In the case of an aggregator creating its own fake devices, we rely on assumption 3 to detect this.

Assumption 8. There is an upper bound, g, on the probability that an honest device goes offline while participating in a round.

This seems plausible given the always-on nature of modern devices, which is being leveraged, e.g., for push notifications. Once a node has decided to participate, we can give a likelihood that it will stay online for a bounded time frame.

Assumption 9. There exists an efficient hash function that is indistinguishable from a random oracle.

We assume that there are hash functions that are sufficiently unpredictable that we can represent them as a random oracle. We use this assumption in two places. The first is in our Algorand-style sortition protocol where, like Algorand does [128], we use the assumption to prove that the sortition is random. This assumption can be avoided in the sortition protocol by replacing the hash functions of signatures by Verifiable Random Functions [213] The second is to enforce non-malleability of ciphertext commitments in the *AGGREGATE* protocol, which prevents $\mathcal{A}$ from committing to ciphertexts that depend on honest devices' ciphertexts. The assumption can be avoided in this protocol by using a non-malleable commitment protocol (e.g., [92]). These protocols tend to require a random public string; B_0 can be used for this.

C.2. Preliminaries

Here we describe some building blocks which are needed by other parts of the protocol.

First off, we rely on the Aggregator, $\mathcal{A}$, signing all of the messages it sends. When we say that a device receives a message from $\mathcal{A}$ and then posts the message to X, this implicitly includes $\mathcal{A}$'s signature. It is also implicit that all messages contain the round number which is publicly known since the rounds occur at regular intervals.

We start off with the protocol for $\mathcal{A}$ to send a message to a device D_i. There exist point-to-point channels between $\mathcal{A}$ and all devices, and for an honest $\mathcal{A}$ this is sufficient. However, to handle a malicious $\mathcal{A}$ we make the communication publicly verifiable when needed. For brevity, later protocols refer to this simply as $\mathcal{A}$ sending a message to D_i.

SEND_MESSAGE

1. D_i waits for a message from $\mathcal{A}$ for the required amount of time. (Since there is a maximum latency network threshold, this is defined.) If D_i receives the expected message from $\mathcal{A}$ in time, then the protocol is complete.
2. Otherwise, D_i posts to X enough information needed to prove that it should receive a message. This is at most the transcript of all messages exchanged between $\mathcal{A}$ and D_i and in practice usually just the last message sent to $\mathcal{A}$ from D_i.
3. $\mathcal{A}$, upon seeing a post from D_i on X, posts to B the message D_i should have received. (If $\mathcal{A}$ doesn't respond at this point, they leave a public record that they deviated from the protocol, and all parties will know they cheated.)
4. D_i reads the message from B.

We extensively use Merkle trees to give devices assurance about data which is stored by the Aggregator but is too large to be stored or checked by any individual device. We use the Merkle tree design detailed in [24]. In brief this allows for proofs that an element is located

at a particular index in an array where the proof size is logarithmic in the array size.

In our case, it is important that devices know the size of the array. Therefore, every time the Aggregator publishes the root of a Merkle tree, they will also publish the size of the underlying array. Also, any time a device checks that an item is part of the array, they also implicitly check that its index is less than the alleged array size.

C.3. Privacy

The privacy of the system depends on a number of other claims about the system, which we demonstrate modularly below.

Claim 1. If a device should receive a message from the Aggregator, they receive it (i.e., protocol *SEND_MESSAGE* works.)

Proof. Essentially, we use the secure point-to-point channels as the default communication means between $\mathcal{A}$ and devices. However, in the case that $\mathcal{A}$ does not send messages that it should, the communication is then forced to take place in a publicly viewable record, where $\mathcal{A}$ must respond or overtly deviate from the protocol. This only happens when either $\mathcal{A}$ or the given device is dishonest.

This works as follows. Say an honest device, d, expects to receive a message through its secure point-to-point channel to $\mathcal{A}$, but it doesn't receive it. The device's knowledge that it should have received a message is determined by the messages that it has sent or received. However, all of this information is public (e.g., $\mathcal{A}$'s broadcasts) except the messages on the point-to-point channel between it and $\mathcal{A}$. It can therefore publish the transcript of such messages to R as evidence that it should receive a message. (In practice, d need only send sufficient messages to prove that it was due to receive a message, which would usually be just the last message it sent $\mathcal{A}$.) This by itself is not sufficient evidence that $\mathcal{A}$ cheated – the device could be trying to frame $\mathcal{A}$. Therefore, $\mathcal{A}$ gets a chance to redeem itself by publishing the message to B. If $\mathcal{A}$ does, then the device gets its message. If $\mathcal{A}$ doesn't, they have left proof that they have cheated. □

Claim 2. If the leader in round i, L_i is honest, then B_{i+1} will be chosen uniformly at random and $\mathcal{A}$ cannot learn anything about B_{i+1} until L_i reveals $\rho_{i,leader}$.

Proof. An honest leader will never have signed $(B_i, i, 2)$ prior to being elected leader. Similarly, $\mathcal{A}$ cannot forge signatures of $(B_i, i, 2)$ except with negligible probability. Therefore, the probability that $\mathcal{A}$ has queried $\eta_{i,\ L_i,\ 2} = \mathsf{sign}_{sk_{leader}}(B_i, i, 2)$ to the random oracle, prior to the leader being announced, is negligible. By the random oracle assumption, $B_{i+1} = \text{Hash}(\eta_{i,\ L_i,\ 2})$ is therefore chosen uniformly at random. □

If the leader is honest in round i, B_{i+1} will be chosen uniformly at random, so $\mathsf{sign}_{sk_j}(B_{i+1}, i, 1)$ will not have been queried to the random oracle by any eligible leader j, so $h_{i+1,j,1}$ will be chosen uniformly at random for each j and all players have an equal probability of becoming leader in round $i + 1$.

As such, any control $\mathcal{A}$ may have of the system is lost if the leader becomes honest. We will now examine, given an initial random B_i which blocks $\mathcal{A}$ can cause the system to reach without an honest node becoming leader. In practice, $\mathcal{A}$ will not know which of their actions will be optimal. However, to provide a lower bound on security and to simplify analysis, we give $\mathcal{A}$ significantly more power than they actually have. We assume that $\mathcal{A}$ is able to determine the values of $h_{i,j,1}$ for all devices. This will allow them to know how choosing a block in one round will determine the leader candidates in the subsequent round. In reality, they only know the distribution of these, so if they cause a particular block to be chosen, they will not be certain how many (if any) of the lowest hashed nodes they will control in the next round.

Let $B_{i,leader}$ be a block generated by an honest leader's signature. Once $B_{i,leader}$ is revealed, $\mathcal{A}$ can construct a tree T_i of all possible blocks it can reach without having an honest leader. Children are recursively defined in this tree as follows. Block $B_{r,l}$ has a child block $B_{r+1,j}$ if $\mathcal{A}$ controls j and, given $B_{r,l}$ as the block from round r, $\mathcal{A}$ can make j the leader in round r with $B_{r+1,j}$ being the resulting block. Additionally, $B_{r,l}$ has a child block $B_{r+1,\perp}$ if $\mathcal{A}$ can

force the default block to be chosen (which happens if $\mathcal{A}$ controls the leader in round r).

Let us make some observations about this tree.

In the case where there is a child $B_{i+1,j} = \text{Hash}(\mathsf{sign}_{sk_j}(B_{i,leader}, i, 2)$, at the point when $B_{i,leader}$ is revealed, $\mathcal{A}$ will not have queried $\mathsf{sign}_{sk_j}(B_{i,leader}, i, 2)$ to the random oracle before (except with negligible probability). Therefore, the random oracle will produce a uniformly random value for $B_{i+1,j}$. Similarly, at the time $B_{i,leader}$ is revealed, since it is randomly chosen from $\{0,1\}^\lambda$ the probability that $(B_{i,leader}, i)$ was queried to the random oracle is also negligible. Therefore, $B_{i+1,\perp}$, if it exists, is also chosen uniformly at random. By induction, all of the vertices in the tree are chosen uniformly at random.

We will only look at the behavior of the tree for nodes at depth at most κ, which we set to 128 for security. Recall that a device must be registered for at least κ rounds to be eligible for being the leader. If $\mathcal{A}$ is ever able to maintain control of leadership for κ rounds, then they can intentionally choose the keys of new nodes such that they will become leaders κ rounds later. We will now show that the probability $\mathcal{A}$ is able to maintain control of leaders for κ rounds is negligible in κ.

Claim 3. The probability that at the start of round i, $\mathcal{A}$ has controlled the leaders since round $i - \kappa$ is less than $i2^{-\kappa}$. when $f \leq 0.2$.

Proof. Let us imagine a counter c, that begins at 0 and increments every time an honest leader is selected. Let $round(c)$ be the round that the honest leader count became c. We have $round(0) = 0$. Clearly, also $c \leq round(c)$, since there can be at most one honest leader per round. If $\mathcal{A}$ could ever maintain permanent control over the leader, leaving the counter at c, then we say $round(c+1) = \infty$.

Intuitively, each time c increments, $\mathcal{A}$ obtains a new random tree $T_{round(c)}$, of reachable blocks that have dishonest leaders, rooted at $B_{round(c),leader_{round(c)}}$. $\mathcal{A}$ will be able to maintain leadership as long as they can using this tree. Once they are forced to allow an honest node

into leadership again, c increments and they get a new random tree.

We know that $B_{round(c)}$ is selected uniformly at random, either because the leader was honest in $round(c)$, or because $c = 0$ and B_0 is selected uniformly at random by some trusted source. Furthermore, all candidate nodes for round $r < round(c) + \kappa$ will have been selected prior to $\mathcal{A}$ learning anything about $B_{round(c)}$ and therefore, before learning anything about B_r.

Let us determine the expected number of children of some node $B_{r,l}$, with $r < round(c)+\kappa$. In the case where $\mathcal{A}$ does not control the top candidate, then there is no way $\mathcal{A}$ can stop the next leader from being honest, so the node has 0 children. The child $B_{r+1,\perp}$ will exist if and only if the top candidate is controlled by $\mathcal{A}$. Since $\mathcal{A}$ controls a portion f of the population, and the leader is chosen at random, the probability of child $B_{r+1,\perp}$ existing is f. Let j_t be the t^{th} top candidate. If the top t candidates are all controlled by $\mathcal{A}$, then B_{r+1,j_t} is a child of $B_{r,l}$. The probability of this occurring is at most f^t. (These are all clearly not independent events; our analysis will be aware of this fact.)

We now calculate the expected number of nodes in the tree at each round r, $E(T, r)$. There is exactly one root, and it is always there, so $E(T, round(c)) = 1$. For greater depths:

$$E(T, round(c) + 1) = f + f + f^2 + f^3 + \cdots = f + \frac{f}{1-f}$$

$$= \frac{2f - f^2}{1-f}$$

And in general, for $r \geq 1$:

$$E(T, round(c) + d) = (f + f + f^2 + f^3 + \cdots)E(T, round(c) + d - 1)$$

$$E(T, round(c) + d) = \left(\frac{2f - f^2}{1-f}\right)^d$$

Therefore, the expected number of states reachable by $\mathcal{A}$ at round $round(c)+\kappa$ is $\left(\frac{2f-f^2}{1-f}\right)^\kappa$.

For $f \leq 0.2$, $\frac{2f-f^2}{1-f} < \frac{1}{2}$. Then the expected value is at most $2^{-\kappa}$. By Markov's inequality, that means the probability that there exists some node at this depth is at most $2^{-\kappa}$.

Therefore, the probability that $round(c+1) \geq round(c) + \kappa$ is less than $2^{-\kappa}$.

Therefore, for any c, the probability, that there exists $0 \leq \bar{c} < c$ such that $round(\bar{c}+1) \geq round(\bar{c}) + \kappa$ is less than $c2^{-\kappa}$. Since $round(c) \geq c$, for any round $i = round(c)$ the probability that $\mathcal{A}$ has controlled a leader for at most κ rounds is at most $i2^{-\kappa}$.

□

Claim 4. If $f \leq 0.2$, the probability that over m rounds the committee ever has at least $t = \frac{2}{5}C$ malicious committee members is upper-bounded by $2mp + negl(\kappa)$, where $p \leq e^{-fC}\left(\frac{5ef}{2}\right)^{\frac{2C}{5}}$

Proof. Let p be the probability that a randomly chosen committee contains more than t malicious members.

First we show that any adversarial behavior by $\mathcal{A}$ can only increase their chance of getting a malicious committee by a constant factor. Formally, for each honest-leader count c, with $i = round(c)$, with a tree T_i of reachable blocks rooted at $B_{i,leader_i}$, where $leader_i$ is honest, the expected number of reachable blocks that would have a malicious committee is $O(p)$.

From Claim 3, the leader will be randomly chosen for all $i \leq r < i + \kappa$ and the tree, T, will be of depth at most κ, (except with negligible probability, $O(r2^{-\kappa})$). The probabilities of each child existing for a node will therefore be the same as in the proof of Claim 3

We know that each child has the same probability distribution except that, since we are assuming the depth is at most κ, each child will have the additional restriction that the maximum depth of its sub-tree must be at most one less than that of its parent.

Let $E(T)$ be the expected number of nodes in this tree. We then have:

$$E(T) \leq 1 + \sum_{child \in children} P(child)E(T)$$

$$E(T) \leq 1 + E(T)\left(f + \sum_{i}^{\infty} f^i\right)$$

$$E(T) \leq 1 + E(T)\left(\frac{2f - f^2}{1 - f}\right)$$

$$E(T)\left(1 - \frac{2f - f^2}{1 - f}\right) \leq 1$$

$$E(T) \leq \frac{1 - f}{1 - 3f + f^2}$$

For $f \leq 0.2$, $E(T) < 2$.

Since each block in the tree is selected uniformly at random at some point (having never been selected prior to this) the expected number of blocks that would produce a malicious committee is less than $2p$.

Therefore, over m rounds, there will be at most m increments of the honest-leader count, so the expected number of malicious committees in any tree is at most $2mp$. By Markov's inequality, the probability that there exists any block at any point over m rounds, reachable by $\mathcal{A}$, that would produce a malicious committee is at most $2mp$ (plus some negligible probability).

Now let us calculate p.

For a uniform random block, the committee will be a uniformly random subset of the population. Since $C \ll N$, drawing players from N does not significantly change the portion of malicious nodes in the remaining pool. Thus, we can approximate the problem by saying that each player has a probability f of being malicious and that these events are independent. We can therefore use Chernoff bounds to limit the probability that t are

malicious.

We have that the expected number of malicious committee members is fC, where C is the size of the committee. Let X_i be a random variable that is 1 if committee member i is malicious, and 0 otherwise. Let $X = \sum_{i=1}^{C} X_i$ be the random variable representing the distribution of the number of malicious committee members. Let μ as the expected value of X. We know $\mu = fC$. Let t be chosen such that iff $\geq t$ members of the committee are malicious, they are able to access the secret key. If we set $t = \frac{2}{5}C$, Chernoff bounds state that:

$p = Pr(X \geq t) \leq \left(\frac{e^{\frac{2}{5f}-1}}{\frac{2}{5f}^{\frac{2}{5f}}} \right)^{fC}$, which simplifies to

$$p \leq e^{-fC} \left(\frac{5ef}{2} \right)^{\frac{2C}{5}}$$

The theorem follows directly. Graphs of the probability of too many nodes becoming malicious for different values of f and C are shown in Figure 5(a).

□

Claim 5. If there are fewer than $t = \frac{2C}{5}$ colluding members of the committee, no entity is able to reconstruct the secret key generated by KeyGen.

Proof. We run SCALE-MAMBA using a Shamir Sharing Scheme. This scheme has the property that for any subset of the parties with a size below a certain threshold, no information about the secret is revealed. We use $\frac{2C}{5}$ as this threshold. SCALE-MAMBA provides these guarantees on any data identified as secret. □

Claim 6. At the end of protocols *AGGREGATE* and *CHECK_AGGREGATION*, if no device has found malicious activity by $\mathcal{A}$, the sum of the ciphertexts published by $\mathcal{A}$ to B

is correct (with high probability) and no inputs of malicious nodes are dependent on inputs of honest nodes.

Proof. We need to show that the Aggregator cannot weaken the differential privacy guarantee by including a user's input in the summation multiple times (or creating Sybils with inputs related to a target user's input). Next, we argue that this type of tampering by the Aggregator will be caught with high probability during the devices' consistency checks.

First, we look at the case where the leaves of S are all correct, i.e., for $1 \leq i \leq N$, $Commit_i$ appears in Merkle tree M_C and commitment t_i is valid and for $1 \leq i \leq N-1$ $\pi_i < \pi_{i+1}$.

Assume for the sake of contradiction that it is possible for an honest device's ciphertext, c_i, to influence some other ciphertext other than its own. An adversary would therefore need to produce a $t_j = \text{Hash}(r_j||c_i||\pi_j)$ and include t_i in M_C prior to device revealing c_i. Since $\pi_i < \pi_{i+1}$ each leaf contains a unique public key, so $\pi_i \neq \pi_j$. But then $\mathcal{A}$ would need to produce some $t_j = \text{Hash}(r_j||c_j||\pi_j)$, where c_j depends on c_i, but without knowledge of c_i. In the Random Oracle assumption, since no party would ever have queried $r_j||c_j||\pi_j$ to the Oracle before, the result of the function will be indistinguishable from random and therefore will not be computable based on any known value, including c_i. Because of this, each device is assured that any other ciphertext that also contains a commitment in the Merkle tree cannot depend on its own ciphertext.

Next we need to show that if $\mathcal{A}$ introduces an error into the leaves of S, she will be caught with high probability.

Let us examine the probability that a particular $j \in [0, N-1]$ is not picked by any honest online device to be v_{init}. For any particular honest online device, the probability that j is *not* picked as v_{init} is $1 - \frac{1}{N}$.

If any $v_{init} \in [i-s+1, i] \bmod N$ is selected by any honest online node, then $\pi_i < \pi_{i+1}$ will be checked. Similarly, if any $v_{init} \in [i-s, i] \bmod N$ is selected by any honest node, then

the other required leaf properties will be checked for leaf i ($Commit_i$ appears in M_C etc). Therefore, the probability that $\mathcal{A}$ can introduce an error into the leaves without a specific honest online node picking any v_{init} that would catch it is $1 - \frac{s}{N}$.

Each honest node is online with probability $1 - g$ and there are at least $(1 - f)N$ honest nodes so the probability that $\mathcal{A}$ introduces an error into the leaves and is not caught is

$$\left(1 - \frac{s(1-g)}{N}\right)^{(1-f)N} \leq e^{-(1-g)(1-f)s}.$$

Next, we need to show that if the summation is not computed correctly, (i.e., $\mathcal{A}$ introduces an error into any non-leaf node of S), then $\mathcal{A}$ will be caught with high probability.

We will deal separately with the cases where the child ciphertexts are leaves and when they are not.

In the former case, a vertex's summation will be checked by an honest online device if both of the vertex's children are in the range $[v_{init}, v_{init} + s] \bmod N$. This will be true of the j^{th} leaf-parent vertex, if the leaves $2j$ and $2j + 1$ are in $[v_{init}, v_{init} + s] \bmod N$. This will occur exactly when $v_{init} \in [2j + 1 - s, 2j]$. For a given online honest device, the probability that this occurs is therefore $1 - \frac{s}{N}$. Again, since devices are online with probability $1 - g$ and there are at least $(1 - f)N$ devices, the probability that $\mathcal{A}$ will not get caught if they introduce an error in the summation of a leaf-parent vertex is $e^{-(1-g)(1-f)s}$.

Finally we look at the probability that $\mathcal{A}$ can introduce an error into a vertex in the summation tree that is at least two generations above the leaves, which we will refer to as grandparents, since they will always be the grandparent of some vertex. There are $\frac{N}{2} - 1$ such vertices. If s is even, then each honest online device will check exactly $\frac{s}{2}$ leaf-parents and therefore will check exactly $\frac{s}{2}$ grandparents. In this case the probability that a specific

grandparent is not checked by a specific honest online device is

$$\left(1 - \frac{1}{\frac{N}{2} - 1}\right)^{\frac{s}{2}} \leq \left(1 - \frac{2}{N}\right)^{\frac{s}{2}} \leq e^{-\frac{s}{N}}$$

If s is odd, then the number of grandparents a device checks depends on v_{max}. If v_{max} is even (which occurs with probability $\frac{1}{2}$), a given online honest device will check $\lceil \frac{s}{2} \rceil$ leaf-parents and $\lfloor \frac{s}{2} \rfloor$ grandparents. If v_{max} is odd, an honest online device will check $\lceil \frac{s}{2} \rceil$ leaf-parents and $\lceil \frac{s}{2} \rceil$ grandparents.

Therefore, the probability that a given grandparent is not checked by a given honest online device is

$$\begin{aligned}
&\frac{1}{2}\left(1 - \frac{1}{\frac{N}{2} - 1}\right)^{\lfloor \frac{s}{2} \rfloor} + \frac{1}{2}\left(1 - \frac{1}{\frac{N}{2} - 1}\right)^{\lceil \frac{s}{2} \rceil} \\
&= \left(1 - \frac{1}{\frac{N}{2} - 1}\right)^{\lfloor \frac{s}{2} \rfloor} \left(\frac{1}{2} + \frac{1}{2}\left(1 - \frac{1}{\frac{N}{2} - 1}\right)\right) \\
&\leq \left(1 - \frac{2}{N}\right)^{\lfloor \frac{s}{2} \rfloor} \left(1 - \frac{1}{N}\right) \\
&\leq e^{-\frac{s}{N}}
\end{aligned}$$

So regardless of whether s is odd or even, the probability that a given grandparent is not checked by a given honest online device is at most $e^{-\frac{s}{N}}$. By the same arguments as above, the probability that an incorrectly added grandparent is not checked by any device is at most $e^{-(1-f)(1-g)s}$.

Therefore, if $\mathcal{A}$ introduces any error into the summation tree, the probability that she will not be caught is at most $e^{-(1-f)(1-g)s}$. This is negligible in s and does not depend on N. For instance, if $f = 0.05$, $g = 0.05$, $s = 5$ is sufficient to have a failure probability of roughly 0.01, and $s = 20$ is sufficient to have a failure probability of less than 10^{-8}.

□

Claim 7. Given a committee with fewer than $t = \frac{2C}{5}$ malicious committee members, the only information $\mathcal{A}$ learns (with high probability) in each round is the result of the differentially-private query.

Proof. The Aggregator receives ciphertexts of the inputs from devices. Since the number of malicious committee members is below the threshold, the threshold secret sharing scheme ensures that the Aggregator gains no information about the secret key. Also, the majority-honest committee will have correctly generated an Additively Homomorphic key pair that is semantically secure. Therefore the ciphertexts leak no information about the underlying plaintexts.

Each device also sends the aggregator a range proof, z, but the zero-knowledge property of the range proof ensures that z leaks no information about the plaintext value, beyond the fact that it is in the required range.

From Claim 6, the input to the committee is computed correctly. The function evaluated is a differentially-private query. The MPC protocol is secure with abort. If the protocol aborts, no information is gained. If it doesn't, then it computes the function result correctly. □

Claim 8. Assuming all committees have fewer than $\frac{2C}{5}$ malicious committee members, the privacy budget limitation will never be violated.

Proof. By induction, the previous certificate contains a correct value for the remaining privacy budget. For the first certificate this is true because the initial privacy budget is public. For all subsequent certificates, this is true because each committee can calculate the budget expended by the query they facilitated and sign on the next certificate the correct remaining budget. If insufficient budget remains to calculate a query, then the honest committee members will refuse to participate, leaving fewer than $\frac{2C}{5}$ participating committee members, which is not enough to sign a new secret key for that round. □

Combining Claims C.4 and C.8 yields the privacy requirement.

C.4. Correctness

We wish to show that the output given to the Aggregator is correct – that is,the correct result of any specific query, with an addition of correctly-specified differentially private noise with parameter ϵ. For this we assume an honest Aggregator. (We feel no obligation to protect a malicious Aggregator.)

As with any other MPC protocol, we cannot prove that the data provided by each device is the truth. We include range proofs to mitigate this problem to prevent rogue devices from entering enormous values to skew the result. But apart from this, by correctness we mean that the protocol outputs the evaluation of the desired function on the inputs the parties provide.

For privacy we had to show that the committee was unlikely to contain $\frac{2C}{5}$ Aggregator-colluding nodes. In this case, we have to show the committee is unlikely to contain $\frac{2C}{5}$ nodes colluding against the Aggregator. The logic follows as before, except the non-colluding nodes have no ability to produce Sybils. Since there is a small portion of such colluding nodes (Assumption 7), the committee maintains an honest majority with high probability. (At least as high as that given for Privacy.) Since the analysis is the same, we omit it here.

If the committee has fewer than $\frac{2C}{5}$ malicious committee members, then the key generation MPC produces a correct keypair that has the additively homomorphic property, or aborts. The Aggregator performs the additions themselves, so is assured that these are correct.

The key reconstruction happens within the MPC. The Shamir secret sharing scheme reconstruction ensures that the key was correctly reconstructed (or if not, the MPC fails for this round). Again, since fewer than $\frac{2}{5}$ of the committee is malicious, the MPC computation for decryption, noising and thresholding is executed correctly (giving a differentially private answer) or aborts.

C.5. Liveness

We show that the Aggregator will continue to be able to perform queries as long as the privacy budget is sufficient. Note that there is a possibility that a query will fail to be performed in a given round if too many of the committee members go offline. However, this event only affects the round in which it occurs, so only delays the Aggregator in performing that query– it does not destroy their ability to perform queries.

First, there will always be a value B_i for every round, because this can always fall back to its default value (namely the hash of the previous block). Hence, there is always a source of randomness to choose a new committee. And finally, any committee that has enough members online will be able to perform key generation and decrypt-noise-threshold protocols.

Now we show the probability of a particular round failing. The probability that any particular committee member goes offline is g, but any malicious node (of proportion up to f)may also go offline. The maximum number of committee members that can go offline without it preventing the committee from completing its task is $\frac{C}{5}$. We assume these events are independent of each other. By Chernoff bounds, the probability of a committee having over $\frac{C}{5}$ offline devices is at most

$$e^{-(f+g)C} \left(5e(f+g)\right)^{\frac{C}{5}}$$

Union bounding over m rounds gives that the probability of this occurring at all over m rounds to be

$$me^{-(f+g)C} \left(5e(f+g)\right)^{\frac{C}{5}}$$

Example values of this are shown in Figure 5 (b).

C.6. Indemnification

We want that the Aggregator cannot be shown to be cheating erroneously. Devices only publish to the reporting framework for two reasons. The first is to request the Aggregator to send a message. In this case the Aggregator can send the message in public, protecting themselves from any accusations. The second reason is to publish any inconsistent claims that the Aggregator has made. If the Aggregator always makes consistent claims, this will never happen. Thus, the Aggregator can only be proven to have cheated if it did in fact cheat.

APPENDIX D : Fuzz for Orchard

Orchard uses the Fuzz functional programming language designed for differential privacy for constructing queries, and applies additional query transformation steps to produce executable code on its MPC framework.

D.1. Basic syntax

Fuzz is a higher-order functional programming language, with additional primitives to compute over private datasets known as "bags", and probabilistic commands that takes a private value, adds noise to it and releases the noised value as public value.

The two most fundamental bag operations are `bmap` (bag map), and `bsum` (bag sum). The operation `bmap` takes a function `f` that transforms each individual element in a bag, and produces a new bag whose values are the outputs of `f` applied to each value in the original bag. The operation `bsum` computes the sum of a bag of numbers, after clipping each number in the bag into some range $[-r, r]$. The clipping is necessary to ensure the differential privacy property.

With `bmap` and `bsum`, we can already perform many useful computations over bags. The function kmeans_iter counts the number of points in a bag by mapping all points in a bag to the value 1, and sums up this with a clip range of $[-1, 1]$.

Another useful bag operation implemented in terms of `bmap` is `bfilter` (bag filter). This operation takes a boolean predicate `f` over values inside a bag, and only keeps elements on which `f` evalutes to True. This is implemented through the following Fuzz code by mapping bag values v into optional values Just v or Nothing based on whether f v is True or False:

```
filter_fun f v =
if f v then Just v else Nothing
bfilter f bag =
bmap (filter_fun f) bag
```

A final bag operation supplied by Fuzz is bpartition (bag partition). The operator bpartition takes a known constant that specifies the number of partitions we are creating, a function that maps each bag value into an integer partition index, and the bag to be partitioned. Bag partition then returns a list of sub-bags, and the ith sub-bag in the list contains all bag values whose partition index evalutes to i. Readers may wonder why bpartition requires a known constant parameter for the partition count, since bpartition can already infer the partition count from the partition index values. This arrangement is required to keep the number of partitions constant, so that the partition count cannot depend on the bag element values, otherwise bag partition may leak private information about the bag values [208].

In addition to bag operations, Fuzz provides probabilistic commands that act as release mechanisms for private values. In this work, we use the Laplace mechanism lap for releasing numerical values, and the exponential mechanism em for releasing categorical values.

The function lap takes a noise width w, and a center c. The value lap c w represents a Laplace distribution centered at c width w, and this distribution can then be sampled from to release a public value from the private center c. The exponential mechanism em takes a list of private scores with length n, and produces a distribution over the integers $[0, n)$. The ith scores supplied to em indicates "preference" or "quality" of the choice i, and the exponential mechanism will select the choice i that approximately maximizes the score, while providing differential privacy protection of the score values.

D.2. Type System

The term-level syntax and runtime characteristics of Fuzz is an ordinary higher-order functional programming language, with 4 primitive bag operators (bmap, bsum, bfilter , and bpartition). However, Fuzz has a unique type system that keeps track of *function sensitivity* and *privacy cost* of programs.

Values in Fuzz are endowed with a distance metric $d(\cdot, \cdot)$. A Fuzz function f with function sensitivity s means if the inputs x_1 and x_2 satisfy $d(x_1, x_2) \leq 1$, then $d(f(x_1), f(x_2)) \leq s$.

Sensitivity analysis is a key ingredient for determining the privacy cost of a program [106]. By statically determining sensitivities of expressions supplied to release mechanisms (lap and em), Fuzz's type system can calculate the privacy cost of running the program in many cases. A notable exception is that Fuzz forbids usage of release mechanisms in unbounded loops. This is because the type system cannot statically determine the number of iterations for such loops, and thus cannot compute the total privacy cost for such loops if they contained release mechanisms.

APPENDIX E : Mycelium Communication Protocol Security

E.1. Path Setup

Want to show that:

1. If the adversary modifies any message or creates its own, this will be detected and trigger a restart of the protocol.

2. Any message drop is detected and triggers a restart of the protocol before any info is leaked to the adversary beyond what can be observed from passive analysis.

3. If there is no message dropped or tampered with through the run of the protocol, then nothing is leaked to the adversary beyond what can be observed from passive analysis.

Together this shows that the protocol leaks no information outside of passive analysis, which we analyze separately in Section 6.5.

Assumptions:

1. AE gives integrity, PEnc and SEnc give confidentiality.

2. A bulletin board can be used to complain about any missed message or violation.

Proof: Claim 1 follows from the properties of AE and PEnc. Next we prove claim 2: If the first message is dropped, this will be detected and reported to the bulletin board. This is because of the C-round MHT and inclusion proof. If a proof is not sent, all honest devices will see the bulletin board complaint and refuse to proceed, restarting the protocol with newly random paths. Note that the detection of this drop may happen after the second hop messages are deposited – this would leak information about the hop if no new paths were selected, but as long as this second hop is not the final hop (triggering a destination key lookup), it leaks no information about the actual edge from source to destination. The

protocol restart means that a new (random) path will be constructed, so the existence of a path from h_1 to h_2 leaks nothing about the topology.

By an induction-like argument, this mechanism will function in the same way for all subsequent intermediate hops (excluding the penultimate hop, which we analyze separately). Up to this point in the protocol, there must have been no previous message drops (or else the protocol would have restarted), and so only honest messages are sent - thus the only leaked information comes from passive analysis. If a message is dropped in this step, then it will be detected and the protocol will restart before the next complete lookup, leaking no additional information. If the source does not receive an ACK from the forwarding device in the relevant round, it will trigger a restart before keys of any destination node are retrieved. This means that no honest devices in this step will continue with the protocol, and so no information about destinations is revealed.

Once h_k receives the request to retrieve the destination's public key, if this hop is malicious, it knows that it's supposed to get the destination's key. So, to avoid malicious nodes from learning the edge between the source and the destination by dropping a message sent from h_k (or h_k itself not sending anything back to the source), after this round, every honest hop checks that the number of messages in their mailbox matches the number of messages they're supposed to receive (and they know this number exactly after this round). In the last step, a message drop simply means that h_1 dropped the message containing the destination's key instead of forwarding it to the source. This leaks no information, but means that the path setup is incomplete - this will also trigger a restart, as before. Therefore, in no step during the protocol is any additional information leaked about the existence of an edge from source to destination.

Claim 3 follows by construction of the protocol. If all messages reach their desired destination, then each round will have an identical number of messages in overall traffic. The cryptographic properties of AE mean that all messages look indistinguishable from random, and so the only information that is leaked relies on malicious hops intersecting their infor-

mation with the global view of the aggregator, and reducing the anonymity sets. This is all passive information. Therefore, our anonymity analysis in Section 6 holds.

All claims hold, so path setup leaks no information beyond that gleaned from the analysis in Section 6.5.

E.2. Forwarding

Assumptions:

1. Every pair of honest nodes (e.g., source and h_1, or h_1 and h_2) can detect a message drop. This is due to the MHT check.

2. AE gives integrity, PEnc and SEnc give confidentiality.

3. There is at least one honest node in a chain.

4. all the paths were established properly, and path establishment leaked no information to the adversary (beyond what can be inferred from passive analysis)

Given these assumptions, we want to show that there is no way for the adversary to modify/drop a message, and change the observable behavior.

Proof: We go through the protocol in order. In the first step, the devices onion-encrypt their messages and deposit them in the mailboxes of their first hops. The outer layer of encryption is using PEnc with the first hop's public key. If they drop any of the first message deposits, this is detected (assumption 1). The path ids are random, assuming that path setup is secure (assumption 4). This first message can be modified by the adversary, but the destination will always eventually detect this modification due to AE (assumption 2).

In each subsequent C-round, we examine the behavior of each forwarding device. Call this h_i. h_i downloads the messages from their mailbox, checks to make sure the aggregator or the previous hop did not drop any messages using assumption 1, removes one encryption layer if possible, and mixes them. If a message has been dropped or h_i is unable to remove

an encryption layer (meaning that the message has been tampered with), it sends a dummy message encrypted with the public key of the next hop (h_{i+1}). It always knows the path because of the security of the path setup.

They upload either the original message, or an encrypted dummy message (along with the path id) to the mailbox of h_{i+1}. Because of the cryptographic properties of our encryption scheme, h_{i+1} can't distinguish between the cases when h_i sent a dummy message and when h_i forwarded the actual message sent from the source. Although any of these intermediate messages can be modified, the destination will always eventually detect this modification due to AE (assumption 2).

This process continues until messages reach the final destination or until all messages have been dropped. The invariant that we guarantee is that at any step, the only valid messages that any intermediate forwarding device holds are either the original messages (with varying levels of encryption stripped off), or dummy messages encrypted with the correct path, and that the total amount of messages sent by any honest device will be constant. This is true at the start of the protocol because we assume the source is honest - otherwise we don't care about protecting their privacy. Each honest forwarding step maintains this invariant since messages are either passed on or replaced with dummy messages. In the very last step, messages may be dropped without replacement by dummy messages, but this does not violate the invariant.

If, by the end of the chain, there is a nonzero number of messages that have not been dropped or tampered with, the destination will be able to decrypt the original message (this is guaranteed by assumption 3 that we have a complete honest chain). Even if this is not the case, assumption 3 guarantees that no information is revealed about the existence of an edge from the source to the destination. This follows from our invariant and because all messages are indistinguishable from random. Since each device will receive a constant number of (seemingly random) messages from any honest device, no malicious forwarding device can learn anything, outside of the identities of the hops it receives and sends messages

to, respectively (and, if there was a chain of consecutive malicious nodes, that entire chain of hops). If there was a complete malicious chain, this knowledge would be enough to completely reconstruct the edge. However, the existence of even one honest forwarding device breaks this chain. Since path selection is random, knowing that a message passes through a given node leaks no additional information about the existence of an edge to either the source or the destination.

APPENDIX F : Arboretum Security

F.0.1 Exponential Mechanism Truncation Protocol

Definition 1 (*Truncated Mechanism).* Given an exponential mechanism satisfying (ϵ_0, 0)-DP, denote $P(o)$ as its probability mass over data item o. Let $0 < L < 1$ be some cutoff such that there exists at least one item with $P(o) \geq L$, and let P_L be the sum of $P(o)$ where $P(o) < L$. Then a Truncated Mechanism returns item o with probability:

$$P^*(o) \coloneqq \begin{cases} \frac{P(o)}{1-P_L}, & \text{if } P(o) \geq L \\ 0, & \text{otherwise} \end{cases}$$

Lemma 9. We can cap the maximum difference in output probability between the original and truncated mechanism for any possible set of outputs, by the cutoff value L multiplied by the number of categories k. That is, no output from the truncated mechanism can differ too much. from the original. More precisely, given O as the set of all possible outputs, $\forall \sigma \subseteq O, |P(\sigma) - P^*(\sigma)| \leq k \cdot L$

Proof. First denote

$$\sigma^+ \coloneqq \{\forall o \in \sigma | P(o) \geq L\}$$

$$\sigma^- \coloneqq \{\forall o \in \sigma | P(o) < L\}$$

We claim $\forall \sigma \subseteq O, |P(\sigma) - P^*(\sigma)| \leq P_L$. To show this, we expand:

$$|P(\sigma) - P^*(\sigma)| = |(P(\sigma^+) - P^*(\sigma^+)) + (P(\sigma^-) - P^*(\sigma^-))|$$

Notice:

$$P(\sigma^+) - P^*(\sigma^+) \leq 0$$

$$P(\sigma^-) - P^*(\sigma^-) \geq 0$$

Therefore, by the property of absolute value, we have:

$$|P(\sigma) - P^*(\sigma)| \leq \max\left(|P(\sigma^+) - P^*(\sigma^+)|, |P(\sigma^-) - P^*(\sigma^-)|\right)$$

Since we are guaranteed the following two properties,

$$|P(\sigma^+) - P^*(\sigma^+)| \leq |P(O^+) - P^*(O^+)| = P_L$$

$$|P(\sigma^-) - P^*(\sigma^-)| \leq |P(O^-) - P^*(O^-)| = P_L$$

Taking max () on both inequalities, we have:

$$\max\left(|P(\sigma^+) - P^*(\sigma^+)|, |P(\sigma^-) - P^*(\sigma^-)|\right) \leq P_L$$

Hence,

$$\begin{aligned} |P(\sigma) - P^*(\sigma)| &\leq \max\left(|P(\sigma^+) - P^*(\sigma^+)|, |P(\sigma^-) - P^*(\sigma^-)|\right) \\ &\leq P_L \end{aligned}$$

Further,

$$P_L \leq \Sigma_{\{o|P(o)<L\}} C \leq \Sigma_{\{o|o\in O\}} L = k \cdot L$$

Therefore,

$$|P(\sigma) - P^*(\sigma)| \leq k \cdot L$$

□

Theorem: *Truncated Mechanism* satisfies (ϵ_0, δ)-DP for $\delta = (e^{\epsilon_0} + 1) \cdot k \cdot L$, where k is the number of items in total.

Proof. Let the exponential mechanism for any two neighboring datasets X, X' have probability mass P and Q respectively. The weight of their cutoff regions are P_L and Q_L respectively, and their corresponding truncated mechanisms have probability distribution P^*, Q^*.

Because the original exponential mechanism is $(\epsilon_0, 0)$-DP, by definition we have $\forall \sigma \subseteq O, P(\sigma) \leq e^{\epsilon_0} Q(\sigma)$. According to Lemma A.1., we have:

$$P^*(\sigma) - k \cdot L \leq P(\sigma) \leq e^{\epsilon_0} \cdot Q(\sigma) \leq e^{\epsilon_0} \cdot (Q^*(\sigma) + k \cdot L)$$

Rearranging the above inequality, we see that:

$$P^*(\sigma) \leq e^{\epsilon_0} \cdot Q^*(\sigma) + (e^{\epsilon_0} + 1) \cdot k \cdot L$$

Therefore, as claimed, Truncated Mechanism is (ϵ_0, δ) DP with $\delta = (e^{\epsilon_0} + 1) \cdot k \cdot L$

□

F.1. Median Algorithm Modification

We present both algorithms, the original and modified versions, and show their equivalence with respect to differential privacy. The key algorithmic difference comes in the number of iterations – while Alg. 1 iterates over all data points (of which, in our setting, can be on the order of a billion) to compute quality scores, Alg. 2 performs the same operations by iterating over all *categories* (for our benchmarks, we used $2^{15} = 32,768$ of these). This improves the algorithm's performance under secure computation, eliminating unecessary operations on data points falling into the same categories. Alg. 2 is also a more natural fit for categorical data, as opposed to Alg. 1, which treats data points as singular values instead of one-hot encoded vectors.

Algorithm 1 The original (unmodified) algorithm 1 for differentially private median computation [12]

Data z, Privacy Params ϵ, Input n, r_l, r_u
Sort z in increasing order
Clip z to range $[r_l, r_u]$
Insert r_l and r_u into z and set $n = n + 2$
Set $maxNoisyScore = -\infty$
Set $argMaxNoisyScore = -1$
for $i \in [1, n)$ **do**
  $logIntervalLength = \log z[i] - z[i-1]$
  $distFromMedian = \left\lceil |i - \frac{n}{2}| \right\rceil$
  $score = logIntervalLength - \frac{\epsilon}{2} \cdot distFromMedian$
  $N \sim Gumbel(0, 1)$
  $noisyScore = score + N$
  if $noisyScore > maxNoisyScore$ **then**
    $maxNoisyScore \leftarrow noisyScore$
    $argMaxNoisyScore \leftarrow i$
$left = z[argMaxNoisyScore - 1]$
$right = z[argMaxNoisyScore]$
$\tilde{m} \sim Unif\ [left, right]$
return $\tilde{m}$

Sketch of Equivalence: We want to show that the pre-noised scores in our modified algorithm (Alg. 2) are equivalent to the pre-noised scores in the original algorithm (Alg. 1). Both algorithms can be theoretically implemented in Arboretum, but a relatively simple argument allows us to make the performance feasible at large scales by eliminating unecessary

Algorithm 2 Our modified algorithm for median computation over one-hot encoded categorical data

Data z, Privacy Params ϵ, Input n, r_l, r_u, C
Each element of z is an array of size C, already sorted by increasing order of category
Clip z to range $[r_l, r_u]$
Insert r_l and r_u into z and set $n = n + 2$
$sums = [0, \ldots, 0]$ $//C$ categories
for $d \in [0, n)$ **do**
  $sums = sums + z[d]$
$nonZeroCategories = []$
$cumulativeSums = []$
for $k \in [0, C)$ **do**
  if $sums[k]! = 0$ **then**
    $nonZeroCategories.append(sums[k])$
    $cumulativeSums.append(cumulativeSums[k-1]$
    $+sums[k])$
$scores = []$
for $k \in [1, C]$ **do**
  $score = \log(nonZeroCategories[k]$
  $-nonZeroCategories[k-1])$
  $distFromMedian = \left\lceil |cumulativeSums[k-1] - \frac{n}{2} + 1| \right\rceil$
  $score = score - \frac{\epsilon}{2} \cdot distFromMedian$
  $scores.append(score)$
for $k \in [0, length(scores))$ **do**
  $N \sim Gumbel(0, 1)$
  $scores[k] = scores[k] + N$
$idx = argmax(scores)$
$left = nonZeroCategories[idx] - 1$
$right = nonZeroCategories[idx]$
$\tilde{m} \sim Unif\,[left, right]$
return $\tilde{m}$

operations.

To prove equivalence, we show that a) each non-zero score generated by 1 is also generated by 2, and b) no additional scores are generated by 2.

We go through all possible non-zero scores generated by 1. First, we notice that for any two consecutive equivalent user values ($z[i] = z[i-1]$ in the sorted vector z), the corresponding $logIntervalLength = -\infty$. As a result, the noisy max will not be affected by

this *logIntervalLength*, and so this corresponding score is zero. This implies that the only *logIntervalLength* values which affect the final scores are those where $z[i]! = z[i-1]$. For each unique value with a non-zero score processed by algorithm 1, we proceed in ascending order. Algorithm 2 contains these same unique values, also sorted in ascending order because of its one-hot encoding approach. For each i in algorithm 1, we use cumulative sums in algorithm 2 to compute the relevant index i, and use the mapping from ciphertext bins to compute $z[i]$. The resulting computation of the score (before noising), is identical, since it relies on the same distFromMedian and logIntervalLength. Therefore, each non-zero score generated by 1 is also generated by 2. The only scores we generate in algorithm 2 are those represented by consecutive pairs of non-equal values that are present in the user dataset. All of these scores are also generated by algorithm 1, since algorithm 1 simply generates a score for every user data point, without skipping any element. Therefore, there can be no score generated by 2 that is not also generated by 1.

Because the pre-noised scores in 2 are equivalent to the pre-noised scores in 1, and we use the same randomized mechanism to introduce noise to these scores, we can conclude that the distribution of these two randomized algorithms are identical. Thus, in particular they have the same differential privacy properties, so any DP guarantees held by 1 are also held by 2.

F.2. Secrecy of the Sample Mechanism

Our secrecy of the sample mechanism has two desired goals:

- **Secrecy:** No participant within or outside of the system can infer which participants' data was included or excluded from the sample.
- **Sampling Accuracy:** We (on expectation) sample roughly a proportion ϕ of all users who participate in the data collection process.

Secrecy: Our security sketch is as follows: for security, the number of bins in all ciphertexts will always be a power of two. The largest parameters we evaluated result in ciphertexts

with $32,768$ total bins. Our protocol thus allows for any proportion of the form $\phi = \frac{x}{32768}$ to be sampled, where x is any integer $\in [1, 32768]$ (In fact, we can also extend this protocol, using multiple ciphertexts to allow additional denominators, but for now we consider just one). Let's consider such arbitrary x. Our analysis will similarly hold for any number of bins $2^m, m \leq 15$, but simply allows for a smaller range of proportions ϕ to be expressed.

The initial step requires uniformly sampling a value j from the range $[1, 32768]$ inside of a secure MPC by one designated committee. Security of this operation reduces to the security of our MP-SPDZ MPC implementation - much like in generating random Laplace noise for differential privacy purposes, we guarantee that no individual party can influence the result of this random selection. This random selection j will be secret-shared between parties, with the property that no individual learns anything about j, and a majority of parties needs to re-combine their shares for j's recovery.

The next step requires devices to uniformly sample an index i, and place their encrypted local input in the i-th bin. Our ZK proofs ensure the validity of this one-hot encoding, by forcing devices to prove (with zero knowledge about their underlying inputs) that only one bin contains a non-zero value. Devices with incorrect proofs will be excluded from the computation. Of course, some small percentage (up to 5%) of devices may submit correctly formatted ciphertexts, but place their values in the wrong bins - we have no way of enforcing this in our protocol, but this does not affect secrecy of sample of the honest devices (nor, in fact, the malicious devices). The resulting ciphertexts will all look indistinguishable from random – specifically, for any pair $a, b \in [1, 32768], a \neq b$, if a device places their values in bin a or bin b, an adversary will have a negligible statistical advantage in distinguishing between the two scenarios, as a result of security of our HE schemes. This gives us secrecy of bin placement for all individuals.

Finally, we proceed to securely aggregate all device ciphertexts. There are two approaches for aggregation - either the original sampling committee recovers j, or they use the VSR protocol [136] to redistribute their shares to a subsequent committee. In the latter case,

secrecy of these shares rests on the security of the VSR protocol, and we linearly compose the probabilities of failure of the committees (in Arboretum, we cap the number of committees' participation to ensure that no committee fails in *the entirety of the system run* with probability 10^{-9}).

For either scenario, the committee, inside of a secure MPC, receives one aggregate ciphertext, encoding the sums of all individual's ciphertexts. This ciphertext is a public input into the MPC. The MPC circuit then does the following: it first decrypts each bin of the ciphertext, and then linearly runs through all bins, performing a secure comparison to determine whether each index is in the range $[j, j + x]$ (modulo 32768). It adds up the bins for all such elements, including 0's for all elements falling outside of the range. The final sum is noised for differential privacy, and then committee members combine their secrete shares so the noised sum can become public and released to the aggregator. Through this process, no member of the committee sees any of the data in the clear, so our security rests on that of the MPC.

The final result received by the aggregator is certified as differentially private, and contains no identifying information about the sampled parties, outside of any inference that the aggregator can attempt to guess from the output about the underlying data distribution. We note that this sort of inference, on an individual sampling level, is explicitly protected by differential privacy. Therefore, we guarantee that our secrecy of the sample protocol introduces no additional information about which users are sampled, outside of inferences that may already be possible in the given data distribution.

Sampling Accuracy: In Arboretum, we set the malicious device fraction f as a small percentage between 1% and 5%, and we use a ciphertext size of 2^{15}.

We provide the following guarantee: on expectation, given a desired sampling fraction ϕ, and under the presence of up to a fraction of f malicious devices, the proportion of sampled

users (out of devices that submit data) will be *at most*

$$\phi + \phi * f,$$

regardless of the malicious strategy of these devices.

We also guarantee that in the *worst case*, the proportion of devices sampled will be $\phi + f$, including all malicious devices. However, we guarantee that this happens with at most probability ϕ. We notice that the effect of this potential skew scales with ϕ – if we set, say, $\phi = 0.5$, then a small skew of malicious devices is more likely, but has less effect on the overall fraction (we are guaranteed a sampling proportion of at most 0.525.) If we set $\phi = 0.01$, then a small skew may have a larger effect, but will only happen with maximum probability 0.01.

Notice that we do not consider sampling *less* than ϕ devices in the presence of malicious users – this is possible, but is actually likely a more desirable outcome, because the sampled dataset may now include fewer inaccurate or misleading results. We focus on proving this upper bound on expectation.

Proof: In the presence of no malicious devices, the properties of uniformly random sampling guarantee that, regardless of our choice $\phi = \frac{x}{b}$, the expectation of the number of devices that will place themselves in each bin is exactly $\frac{1}{b}$. This means that, in an interval of x determined by the committee, we will sample $x \cdot \frac{1}{b}$ devices, giving us an expected fraction of $\frac{x}{b} = \phi$ devices sampled.

However, in the presence of $f \cdot D$ malicious devices, where D is the total number of participating devices, we note that these devices can, in the worst case, place themselves in any arbitrary bin. First we handle the worst case - when malicious devices can somehow guess the correct interval, and collude to place all of themselves in that bin. Because the interval is randomly selected and hidden from all devices, and is exactly a ϕ-proportion of the 2^m bins, this result (giving a total proportion of at most $\phi + f$) can happen with at

most probability ϕ, as claimed above.

When calculating the expected value, we first consider this strategy of collusion to place all malicious devices in one bin or set of bins, to maximize the set of malicious devices sampled. This means that the true proportion P of devices sampled under this malicious strategy will be $\phi + f$ with probability ϕ, and $\leq \phi$ with probability $1 - \phi$. Hence, we see that:

$$E[P] \leq (\phi + f) \cdot \phi + \phi(1 - \phi) = \phi^2 + f \cdot \phi + \phi - \phi^2 = \phi + \phi * f,$$

as claimed.

Notice that *all other malicious strategies* result in an expectation less than or equal to one shown above, because no single device can have a strategy that places it in the correct bin with probability more than ϕ. Therefore, as claimed above, the proportion of sampled users will on expectation be *at most*

$$\phi + \phi * f.$$

F.3. Security analysis

As described in Section 7.1, Arboretum has four goals: accuracy, integrity, privacy, and efficiency. Arboretum is accurate because (unlike, say, LDP [113]), it does not add any extra perturbation beyond the minimum that is required for differential privacy in the central model. Integrity is a result of our two verification protocols – ZK proofs for malicious participants, and MHT verification for a malicious aggregator. The ZK proofs guarantee that malicious participants cannot corrupt the results by submitting malformed inputs – say, by pretending that their user is 1,000 years old. Of course they can skew the results by providing well-formed but incorrect inputs, but the effect should be roughly proportional to the fraction of users that are malicious, which we have assumed to be small (Section 7.1).

Privacy in Arboretum boils down to two components: output privacy and individual input privacy. For output privacy, by only releasing the final result of any query once it has

been securely perturbed by a committee, we guarantee (ϵ, δ)-differential privacy, provided that we have no MPC failure (the analysis is analogous to that in Appendix C) and that our cryptographic assumptions hold (see Section 7.4). For individual input privacy, we guarantee, based on the computational hardness of the RLWE problem, that no individual's input can be revealed to any other participant within or outside of the system (including other participants, committee member, or the central aggregator). This guarantee comes as a result of input data never leaving any individual device unless it has been encrypted under one of our HE schemes, and never being released in the clear unless it has been aggregated and appropriately noised.

With the exception of our new secrecy of the sample mechanism (F.2), we note that all of our security mechanisms in Arboretum have been adapted from prior work, and that the original security proofs continue to apply. Our sortition mechanism has been proved bias-resistant in [128] and used securely in Honeycrisp. The latter also contains proofs of security for our MHT inclusion proofs. Security of of our VSR protocol to distribute shares between committees is discussed in Chapter 6.

The only additional security implication for Arboretum is that it has the ability to potentially use many committees for a single round. However, this simply increases the probability of failure linearly in the number of total committees per query. We note that to keep the probability of failure extremely low, Arboretum generates a new keypair after each query. As it stands, a query plan generated by Arboretum could use, say, 1000 committees in a single query and have a probability of failure smaller than $2 \cdot 10^{-9}$. Therefore, we refresh the keypair, guaranteeing privacy for all previous queries by retiring the previous secret keys that their security relies on.

F.4. Accuracy of the cost model

The costs of running queries (Section 6.3) are predictions by Arboretum, using our cost model from Section 7.2.7, so their accuracy depends on the accuracy of the model. To quantify this accuracy, we picked 15 query plans – the ten best ones chosen by Arboretum

for our queries, and 5 random plans that were not chosen – and measured the cost of executing unique vignette types from these plans on actual reference hardware (that is, the hardware we used to derive the cost model). We then compared the per-vignette bandwidth and computation costs to the actual costs. For the set of 26 unique vignettes, Arboretum's estimates were within 5% of the actual bandwidth for half of the vignettes, and within 25% of the actual bandwidth for all except one vignette. Computation cost estimates were within 10% of the actual compute time for half of the vignettes, and within 33% of the actual compute time for all except 3 of the vignettes.

Our cost model was based on benchmarking individual MPC building blocks with committees of size 40, but since Arboretum updates the committee size based on the total number of committees while evaluating a given plan, the bandwidth and timing costs have some skew when the committee size is updated. The compute costs for the MPC vignettes have more variance because, unlike bandwidth costs, the compute costs don't scale linearly with the size of the committee in MP-SPDZ. For the 4 outliers (1 for bandwidth and 3 for compute cost estimates), the estimates were roughly twice the actual costs. This was due to the fact that since Arboretum does not account for all possible MPC optimizations, it overestimates the costs of some vignettes.

BIBLIOGRAPHY

[1] Exposure notifications: Using technology to help public health authorities fight COVID-19. `https://www.google.com/covid19/exposurenotifications/`.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Whitepaper; software available from tensorflow.org.

[3] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[4] E. A. Abbe, A. E. Khandani, and A. W. Lo. Privacy-preserving methods for sharing financial risk exposures. *American Economic Review*, 102(3):65–70, 2012.

[5] J. Abowd, R. Ashmead, G. Simson, D. Kifer, P. Leclerc, A. Machanavajjhala, and W. Sexton. Census topdown: Differentially private data, incremental schemas, and consistency with public knowledge. *US Census Bureau*, 2019.

[6] J. M. Abowd, R. Ashmead, R. Cumings-Menon, S. Garfinkel, M. Heineck, C. Heiss, R. Johns, D. Kifer, P. Leclerc, A. Machanavajjhala, et al. The 2020 census disclosure avoidance system topdown algorithm. *arXiv preprint arXiv:2204.08986*, 2022.

[7] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (Csur)*, 51(4):1–35, 2018.

[8] G. Ács and C. Castelluccia. I have a dream! (differentially private smart metering). In *Information Hiding*, 2011.

[9] D. Adam, P. Wu, J. Wong, E. Lau, T. Tsang, S. Cauchemez, G. Leung, and B. Cowling. Clustering and superspreading potential of severe acute respiratory syndrome coronavirus 2 (sars-cov-2) infections in hong kong. *Nature Medicine*, 2020.

[10] N. Agarwal, A. T. Suresh, F. X. X. Yu, S. Kumar, and B. McMahan. cpsgd: Communication-efficient and differentially-private distributed SGD. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[11] D. Agrawal and D. Kesdogan. Measuring anonymity: The disclosure attack. *IEEE Security & Privacy*, 1(6), Nov. 2003.

[12] D. Alabi, A. McMillan, J. Sarathy, A. Smith, and S. Vadhan. Differentially private simple linear regression. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2020.

[13] A. Albarghouthi and J. Hsu. Synthesizing coupling proofs of differential privacy. *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2017.

[14] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, 2018.

[15] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptography*, 9(3):169–203, 2015.

[16] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[17] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[18] Apple. Apple reports first quarter results. Press release, February 2018; `https://www.apple.com/newsroom/2018/02/apple-reports-first-quarter-results/`.

[19] Apple. Differential privacy. `https://images.apple.com/privacy/docs/Differential_Privacy_Overview.pdf`.

[20] Apple. Reports on government information requests. `https://www.apple.com/legal/transparency/report-pdf.html`.

[21] Apple. A message to our customers. `https://www.apple.com/customer-letter/`, Feb. 2016.

[22] Apple Differential Privacy Team. Learning with privacy at scale. *Apple Machine Learning Journal*, 2017.

[23] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein. Optimized honest-majority mpc for malicious adversaries—breaking the 1 billion-gate per second barrier. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[24] E. K. B. Laurie, A. Langley. Certificate transparency. RFC 6962, RFC Editor, June 2013.

[25] L. Backstrom, C. Dwork, and J. Kleinberg. Wherefore art thou r3579x?: Anonymized social networks, hidden patterns, and structural steganography. In *International World Wide Web Conference (WWW)*, 2007.

[26] E. Bagdasaryan, P. Kairouz, S. Mellem, A. Gascón, K. Bonawitz, D. Estrin, and M. Gruteser. Towards sparse federated analytics: Location heatmaps under distributed differential privacy with secure aggregation. *arXiv preprint arXiv:2111.02356*, 2021.

[27] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii. Scalable k-means++. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2012.

[28] B. Balle, G. Barthe, and M. Gaboardi. Privacy amplification by subsampling: Tight analyses via couplings and divergences. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[29] B. Balle, J. Bell, A. Gascón, and K. Nissim. The privacy blanket of the shuffle model. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2019.

[30] S. Barocas and K. Levy. Privacy dependencies. *Wash. L. Rev.*, 95:555, 2020.

[31] G. Barthe, M. Gaboardi, E. J. Gallego Arias, J. Hsu, A. Roth, and P.-Y. Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2015.

[32] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2013.

[33] J. Bater, G. Elliott, C. Eggen, S. Goel, A. Kho, and J. Rogers. SMCQL: Secure querying for federated databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2017.

[34] D. Beaver. Efficient multiparty protocols using circuit randomization. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1991.

[35] J. H. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova. Secure single-server aggregation with (poly) logarithmic overhead. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.

[36] M. Bellare, R. Ng, and B. Tackmann. Nonces are noticed: AEAD revisited. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2019.

[37] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, 2018:46, 2018.

[38] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable zero knowledge with no trusted setup. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2019.

[39] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. Aurora: Transparent succinct arguments for R1CS, 2019.

[40] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the USENIX Security Symposium*, 2014.

[41] J. D. C. Benaloh. *Verifiable secret-ballot elections*. PhD thesis, Yale University, 1987.

[42] A. Bhowmick, J. Duchi, J. Freudiger, G. Kapoor, and R. Rogers. Protection against reconstruction and its applications in private federated learning. *arXiv:1812.00984 [cs, stat]*, 2018.

[43] Q. Bi, Y. Wu, S. Mei, C. Ye, X. Zou, Z. Zhang, X. Liu, L. Wei, S. A. Truelove, T. Zhang, W. Gao, C. Cheng, X. Tang, X. Wu, Y. Wu, B. Sun, S. Huang, Y. Sun, J. Zhang, T. Ma, J. Lessler, and T. Feng. Epidemiology and transmission of COVID-19 in 391 cases and 1286 of their close contacts in Shenzhen, China: a retrospective cohort study. *Lancet Infectious Diseases*, 2020.

[44] V. Bindschaedler, S. Rane, A. E. Brito, V. Rao, and E. Uzun. Achieving differential privacy in secure multiparty data aggregation protocols on star networks. In *Conference on Data and Application Security and Privacy (CODASPY)*, 2017.

[45] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, 2012.

[46] A. Bittau, U. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnes, and B. Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[47] J. Blocki, A. Blum, A. Datta, and O. Sheffet. Differentially private data analysis of social networks via restricted sensitivity. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, 2013.

[48] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the SuLQ framework. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2005.

[49] D. Bogdanov, L. Kamm, B. Kubo, R. Rebane, V. Sokk, and R. Talviste. Students and taxes: a privacy-preserving study using secure computation. *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2016.

[50] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.

[51] J. Böhler and F. Kerschbaum. Secure multi-party computation of differentially private median. In *Proceedings of the USENIX Security Symposium*, 2020.

[52] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. M. Kiddon, J. Konecny, S. Mazzocchi, B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander. Towards federated learning at scale: System design. In *Proceedings of Machine Learning and Systems*, 2019.

[53] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical Secure Aggregation for Federated Learning on User-Held Data. *arXiv:1611.04482 [cs, stat]*, 2016.

[54] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[55] D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-dnf formulas on ciphertexts. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2005.

[56] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. 2019.

[57] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2011/277, 2011. `https://eprint.iacr.org/2011/277`.

[58] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on computing*, 43(2):831–871, 2014.

[59] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software Grand eXposure: SGX cache attacks are practical. In *Proceedings of the USENIX Workshop on Offensive Technologies*, 2017.

[60] C. Buckler. Average page weight increases 15% in 2014, Dec. 2014. `https://www.sitepoint.com/average-page-weight-increases-15-2014/`.

[61] B. Bunz, J. Bootle, D. Boneh, A. Poelstra, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[62] B. Bünz, B. Fisch, and A. Szepieniec. Transparent snarks from dark compilers. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2020.

[63] M. Burkhart, M. Strasser, D. Many, and X. A. Dimitropoulos. Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. In *Proceedings of the USENIX Security Symposium*, 2010.

[64] R. Canetti, Y. T. Kalai, A. Lysyanskaya, R. L. Rivest, A. Shamir, E. Shen, A. Trachtenberg, M. Varia, and D. J. Weitzner. Privacy-preserving automated exposure notification. *IACR Cryptol. ePrint Arch*, 2020.

[65] C. L. Canonne, G. Kamath, A. McMillan, A. Smith, and J. Ullman. The structure of optimal private tests for simple hypotheses. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, 2019.

[66] X. Cao, J. Jia, and N. Z. Gong. Data poisoning attacks to local differential privacy protocols. In *Proceedings of the USENIX Security Symposium*, 2021.

[67] T.-H. H. Chan, E. Shi, and D. Song. Private and Continual Release of Statistics. In *ACM Transactions on Information and System Security*, 2011.

[68] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 1981.

[69] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards statistical queries over distributed private user data. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[70] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2015.

[71] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2017.

[72] A. Cheu. Differential privacy in the shuffle model: A survey of separations. *arXiv preprint arXiv:2107.11839*, 2021.

[73] A. Cheu, A. Smith, and J. Ullman. Manipulation attacks in local differential privacy. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[74] A. Cheu, A. D. Smith, J. Ullman, D. Zeber, and M. Zhilyaev. Distributed differential privacy via shuffling. *IACR Cryptology ePrint Archive*, 2019:245, 2019.

[75] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2020.

[76] H. Cho, D. J. Wu, and B. Berger. Secure genome-wide association analysis using multiparty computation. *Nature biotechnology*, 36(6):547–551, 2018.

[77] S. Clifford. Web privacy on the radar in congress. *New York Times*, 2008.

[78] G. Cormode, S. Jha, T. Kulkarni, N. Li, D. Srivastava, and T. Wang. Privacy at scale: Local differential privacy in practice. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1655–1658, 2018.

[79] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[80] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2015.

[81] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[82] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty ccmputation from somewhat homomorphic encryption. In *Proceedings of the International Cryptology Conference (CRYPTO)*. 2012.

[83] G. Danezis. Statistical disclosure attacks. In *IFIP International Information Security Conference*, 2003.

[84] G. Danezis, C. Fournet, M. Kohlweiss, and S. Zanella-Béguelin. Smart meter aggregation via secret-sharing. In *Proceedings of the First ACM Workshop on Smart Energy Grid Security*, 2013.

[85] L. Danon, J. M. Read, T. A. House, M. C. Vernon, and M. J. Keeling. Social encounter networks: characterizing great britain. *Proceedings of the Royal Society B: Biological Sciences*, 2013.

[86] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. P. B. authors contributed equally). Gluon: A communication optimizing framework for distributed heterogeneous graph analytics. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[87] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. E. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz. Chet: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019.

[88] A. A. de Amorim, M. Gaboardi, E. J. Gallego Arias, and J. Hsu. Really natural linear indexed type checking. In *International Symposium on Implementation and Application of Functional Languages (IFL)*, 2014.

[89] A. A. de Amorim, M. Gaboardi, J. Hsu, and S. Katsumata. Probabilistic relational reasoning via metrics. In *ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2019.

[90] D. Desfontaines and B. Pejó. SOK: Differential Privacies. *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2020.

[91] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1989.

[92] G. Di Crescenzo, J. Katz, R. Ostrovsky, and A. Smith. Efficient and non-interactive non-malleable commitment. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2001.

[93] M. v. Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2010.

[94] B. Ding, J. Kulkarni, and S. Yekhanin. Collecting telemetry data privately. In *NIPS*, 2017.

[95] Z. Ding, Y. Wang, Y. Xiao, G. Wang, D. Zhang, and D. Kifer. Free gap estimates from the exponential mechanism, sparse vector, noisy max and related algorithms. *VLDB Journal*, Feb. 2022.

[96] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the USENIX Security Symposium*, 2004.

[97] I. Dinur and K. Nissim. Revealing information while preserving privacy. In *PODS*, 2003.

[98] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[99] D. Durfee and R. Rogers. Practical differentially private top-k selection with pay-what-you-get composition. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[100] C. Dwork, A. Karr, K. Nissim, and L. Vilhuber. On privacy in the age of covid-19. *Journal of Privacy and Confidentiality*, 2020.

[101] C. Dwork and J. Lei. Differential privacy and robust statistics. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2009.

[102] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2006.

[103] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum. Differential privacy under continual observation. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2010.

[104] C. Dwork, M. Naor, O. Reingold, G. N. Rothblum, and S. Vadhan. On the complexity of differentially private data release: Efficient algorithms and hardness results. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2009.

[105] C. Dwork and K. Nissim. Privacy-preserving datamining on vertically partitioned databases. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2004.

[106] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[107] C. Dwork, G. N. Rothblum, and S. Vadhan. Boosting and differential privacy. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 2010.

[108] J. Eisenstein. What to do about bad language on the internet. *NAACL-HLT*, 2013(4):359–369, June 2013.

[109] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[110] A. Endo, S. Abbott, A. J. Kucharski, S. Funk, et al. Estimating the overdispersion in covid-19 transmission using outbreak sizes outside china. *Wellcome Open Research*, 2020.

[111] Ú. Erlingsson, V. Feldman, I. Mironov, A. Raghunathan, S. Song, K. Talwar, and A. Thakurta. Encode, shuffle, analyze privacy revisited: Formalizations and empirical evaluation. *arXiv preprint arXiv:2001.03618*, 2020.

[112] U. Erlingsson, V. Feldman, I. Mironov, A. Raghunathan, K. Talwar, and A. Thakurta. Amplification by shuffling: From local to central differential privacy via anonymity. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2019.

[113] U. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: Randomized aggregatable privacy-preservingordinal response. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[114] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012.

[115] G. Fanti, V. Pihur, and U. Erlingsson. Building a RAPPOR with the Unknown: Privacy-Preserving Learning of Associations and Data Dictionaries. *arXiv:1503.01214 [cs]*, 2015.

[116] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer communication across network address translators. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2005.

[117] P. Fox-Penner. A year later, lessons from the blackout. *The New York Times*, Aug. 2004. `https://www.nytimes.com/2004/08/15/opinion/a-year-later-lessons-from-the-blackout.html`.

[118] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2002.

[119] A. Friedman and A. Schuster. Data mining with differential privacy. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Washington, DC, USA, 2010.

[120] K. B. Frikken and P. Golle. Private social network analysis: How to assemble pieces of a graph privately. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, 2006.

[121] D. Froelicher, P. Egger, J. Sousa, J. L. Raisaro, Zhicong Huang, C. Mouchet, B. Ford, and J.-P. Hubaux. UnLynx: A Decentralized System for Privacy-Conscious Data Sharing. *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2017.

[122] D. Froelicher, P. Egger, J. S. Sousa, J. L. Raisaro, Zhicong Huang, C. Mouchet, B. Ford, and J.-P. Hubaux. UnLynx: A Decentralized System for Privacy-Conscious Data Sharing. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2017.

[123] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2013.

[124] S. Garfinkel, J. M. Abowd, and C. Martindale. Understanding database reconstruction attacks on public data. *Communications of the ACM*, 62(3):46–53, 2019.

[125] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2009.

[126] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. In *Annual Cryptology Conference*, pages 850–867. Springer, 2012.

[127] R. C. Geyer, T. Klein, and M. Nabi. Differentially private federated learning: A client level perspective. *arXiv preprint arXiv:1712.07557*, 2017.

[128] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[129] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory. *arXiv preprint arXiv:1904.07162*, 2019.

[130] A. M. Girgis, D. Data, S. Diggavi, A. T. Suresh, and P. Kairouz. On the renyi differential privacy of the shuffle model. In *CCS*, 2021.

[131] S. Goldwasser and S. Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 365–77, 1982.

[132] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1985.

[133] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 18(1):186–208, 1989.

[134] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of*

the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2014.

[135] Google Ads. Building a privacy-first future for web advertising. 2021. `https://blog.google/products/ads-commerce/2021-01-privacy-sandbox/`.

[136] K. Gopinath and V. H. Gupta. An extended verifiable secret redistribution protocol for archival systems. In *Proceedings. The First International Conference on Availability, Reliability and Security*, 2006.

[137] S. Gorbunov, V. Vaikuntanathan, and D. Wichs. Leveled fully homomorphic signatures from standard lattices. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2015.

[138] S. Goryczka and L. Xiong. A comprehensive comparison of multiparty secure additions with differential privacy. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[139] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, page 2. ACM, 2017.

[140] J. Groth. On the size of pairing-based non-interactive arguments. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2016.

[141] D. F. Gudbjartsson, A. Helgason, H. Jonsson, O. T. Magnusson, P. Melsted, G. L. Norddahl, J. Saemundsdottir, A. Sigurdsson, P. Sulem, A. B. Agustsdottir, et al. Spread of sars-cov-2 in the icelandic population. *New England Journal of Medicine*, 2020.

[142] A. Gupta, K. Ligett, F. McSherry, A. Roth, and K. Talwar. Differentially private combinatorial optimization. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010.

[143] A. Gupta, A. Roth, and J. Ullman. Iterative constructions and private data release. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2012.

[144] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proceedings of the USENIX Security Symposium*, 2011.

[145] S. Halevi, Y. Lindell, and B. Pinkas. Secure computation on the web: Computing without simultaneous interaction. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2011.

[146] S. Halevi, Y. Lindell, and B. Pinkas. Secure computation on the web: Computing without simultaneous interaction. IACR ePrint 2011/157, 2011.

[147] S. Halevi and V. Shoup. Design and implementation of a homomorphic-encryption library. *IBM Research (Manuscript)*, 6(12-15):8–36, 2013.

[148] M. Hardt, B. Recht, and Y. Singer. Train faster, generalize better: Stability of stochastic gradient descent. In *International conference on machine learning*, 2016.

[149] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. Sok: General purpose compilers for secure multi-party computation. Proceedings of the IEEE Symposium on Security and Privacy (S&P), 2019.

[150] M. Hay, G. Miklau, D. Jensen, D. Towsley, and P. Weis. Resisting structural re-identification in anonymized social networks. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2008.

[151] M. Hay, V. Rastogi, G. Miklau, and D. Suciu. Boosting the accuracy of differentially private histograms through consistency. *Proceedings of the VLDB Endowment*, 2010.

[152] B. Hemenway, S. Lu, R. Ostrovsky, and W. Welser Iv. High-precision secure computation of satellite collision probabilities. In *International Conference on Security and Cryptography for Networks*, pages 169–187. Springer, 2016.

[153] S. Hohenberger, S. Myers, R. Pass, and a. shelat. ANONIZE: A large-scale anonymous survey system. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014.

[154] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. Zcash protocol specification. 2016.

[155] I. Hoque and I. Gupta. Lfgraph: Simple and fast distributed graph analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, 2013.

[156] J. Hsu, M. Gaboardi, A. Haeberlen, S. Khanna, A. Narayan, B. C. Pierce, and A. Roth. Differential privacy: An economic method for choosing epsilon. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2014.

[157] HTTP Archive. Report: Page weight. `https://httparchive.org/reports/page-weight`, 2020.

[158] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2003.

[159] T. Hulsen. Sharing is caring—data sharing initiatives in healthcare. *International Journal of Environmental Research and Public Health*, 17(9):3046, 2020.

[160] C. Ilvento. Implementing the exponential mechanism with base-2 differential privacy. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.

[161] M. Jawurek and F. Kerschbaum. Fault-tolerant privacy-preserving statistics. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2012.

[162] Q.-L. Jing, M.-J. Liu, Z.-B. Zhang, L.-Q. Fang, J. Yuan, A.-R. Zhang, N. E. Dean, L. Luo, M.-M. Ma, I. Longini, et al. Household secondary attack rate of covid-19 and associated determinants in guangzhou, china: a retrospective cohort study. *The Lancet Infectious Diseases*, 2020.

[163] M. Joseph. *Differential Privacy Beyond the Central Model*. PhD thesis, University of Pennsylvania, 2020.

[164] M. Joye and B. Libert. A Scalable Scheme for Privacy-Preserving Aggregation of Time-Series Data. In *Proceedings of the International Financial Cryptography Conference*, 2013.

[165] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, et al. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*, 2019.

[166] P. Kairouz, S. Oh, and P. Viswanath. The composition theorem for differential privacy. In *International conference on machine learning*. PMLR, 2015.

[167] S. P. Kasiviswanathan and A. Smith. On the'semantics' of differential privacy: A bayesian formulation. *Journal of Privacy and Confidentiality*, 6(1), 2014.

[168] M. Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.

[169] M. Keller, E. Orsini, and P. Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[170] M. Keller, V. Pastro, and D. Rotaru. Overdrive: making SPDZ great again. In *EUROCRYPT*, 2018.

[171] A. Korolova. Privacy violations using microtargeted ads: A case study. In *IEEE International Conference on Data Mining Workshops*, 2010.

[172] A. Kosba, Z. Zhao, A. Miller, Y. Qian, H. Chan, C. Papamanthou, R. Pass, abhi shelat, and E. Shi. C∅C∅: A framework for building composable zero-knowledge proofs. Cryptology ePrint Archive, Report 2015/1093, 2015.

[173] M. Kraitsberg, Y. Lindell, V. Osheter, N. P. Smart, and Y. T. Alaoui. Adding distributed decryption and key generation to a ring-lwe based cca encryption scheme. In *Australasian Conference on Information Security and Privacy*, 2019.

[174] B. Kreuter, A. Shelat, and C.-H. Shen. {Billion-Gate} secure computation with malicious adversaries. In *Proceedings of the USENIX Security Symposium*, 2012.

[175] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the USENIX Security Symposium*, 2012.

[176] KU Leuven COSIC. SCALE-MAMBA. `https://github.com/KULeuven-COSIC/SCALE-MAMBA`.

[177] K. Kursawe, G. Danezis, and M. Kohlweiss. Privacy-friendly aggregation for the smart-grid. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2011.

[178] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2017.

[179] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[180] A. Kwon, D. Lu, and S. Devadas. {XRD}: Scalable messaging system with cryptographic privacy. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[181] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury. Oort: Efficient federated learning via guided participant selection. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[182] M. S. Lam, S. Guo, and J. Seo. Socialite: Datalog extensions for efficient social network analysis. In *Proceedings of the IEEE International Conference on Data Engineering*, 2013.

[183] A. Lapets, F. Jansen, K. D. Albab, R. Issa, L. Qin, M. Varia, and A. Bestavros. Accessible privacy-preserving web-based data analysis for assessing and addressing economic inequalities. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, pages 1–5, 2018.

[184] R. Laxminarayan, B. Wahl, S. R. Dudala, K. Gopal, S. Neelima, K. J. Reddy, J. Radhakrishnan, J. A. Lewnard, et al. Epidemiology and transmission dynamics of covid-19 in two indian states. *Science*, 2020.

[185] D. Lazar, Y. Gilad, and N. Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[186] S. Le Blond, D. Choffnes, W. Zhou, P. Druschel, H. Ballani, and P. Francis. Towards efficient traffic-analysis resistant anonymity networks. *Proceedings of the ACM SIGCOMM Conference*, 2013.

[187] M. Lécuyer, R. Spahn, K. Vodrahalli, R. Geambasu, and D. Hsu. Privacy accounting and quality control in the sage differentially private ml platform. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[188] I. Leontiadis, K. Elkhiyaoui, M. Önen, and R. Molva. PUDA - Privacy and unforgeability for data aggregation. In *International Conference on Cryptology and Network Security (CANS)*, 2015.

[189] J. A. Lewis, D. E. Zheng, and W. A. Carter. *The Effect of Encryption on Lawful Access to Communications and Data: A Report of the Center for Strategic and International Studies (CSIS) Technology Policy Program.* Rowman & Littlefield, 2017.

[190] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *2007 IEEE 23rd international conference on data engineering*, pages 106–115. IEEE, 2007.

[191] D. Lie and P. Maniatis. Glimmers: Resolving the privacy/trust quagmire. *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[192] Y. Lindell. Secure multiparty computation (mpc). Cryptology ePrint Archive, Report 2020/300, 2020. `https://ia.cr/2020/300`.

[193] K. Liu and E. Terzi. Towards identity anonymization on graphs. In *Proceedings of the ACM SIGMOD Conference*, 2008.

[194] Y. Liu, R. M. Eggo, and A. J. Kucharski. Secondary attack rate and superspreading events for sars-cov-2. *The Lancet*, 2020.

[195] J. O. Lloyd-Smith, S. J. Schreiber, P. E. Kopp, and W. M. Getz. Superspreading and the effect of individual variation on disease emergence. *Nature*, 2005.

[196] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *Uncertainty in Artificial Intelligence*, 2010.

[197] T. Luo, M. Pan, P. Tholoniat, A. Cidon, R. Geambasu, and M. Lécuyer. Privacy budget scheduling. In *OSDI*, 2021.

[198] LWE estimator tool. `https://bitbucket.org/malb/lwe-estimator/`, commit 3019847.

[199] M. Lyu, D. Su, and N. Li. Understanding the sparse vector technique for differential privacy. *arXiv preprint arXiv:1603.01699*, 2016.

[200] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2010.

[201] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):3–es, 2007.

[202] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD Conference*, 2010.

[203] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C.Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD Conference*, 2010.

[204] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updateable structured reference strings. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[205] S. Mazloom and S. D. Gordon. Secure computation with differentially private access patterns. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[206] F. McSherry and R. Mahajan. Differentially-private network trace analysis. In *Proceedings of the ACM SIGCOMM Conference*, 2010.

[207] F. McSherry and K. Talwar. Mechanism design via differential privacy. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 2007.

[208] F. D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proceedings of the ACM SIGMOD Conference*, 2009.

[209] L. Melis, G. Danezis, and E. De Cristofaro. Efficient Private Statistics with Succinct Sketches. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[210] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[211] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1987.

[212] S. Micali, O. Goldreich, and A. Wigderson. How to play any mental game. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1987.

[213] S. Micali, M. Rabin, and S. Vadhan. Verifiable random functions. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999.

[214] D. Mir, S. Muthukrishnan, A. Nikolov, and R. N. Wright. Pan-private algorithms via statistics on sketches. In *Proceedings of the ACM SIGMOD Conference*, 2011.

[215] I. Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[216] I. Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[217] P. Mittal, M. Wright, and N. Borisov. Pisces: Anonymous communication using social networks. *arXiv preprint arXiv:1208.6326*, 2012.

[218] A. Moghimi, G. Irazoqui, and T. Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.

[219] I. Morris. Apple Has Sold 1.2 Billion iPhones Worth $738 Billion In 10 Years. *Forbes*. June 29, 2017. `https://www.forbes.com/sites/ianmorris/2017/06/29/apple-has-sold-1-2-billion-iphones-worth- 738-billion-in-10-years/`.

[220] J. Mossong, N. Hens, M. Jit, P. Beutels, K. Auranen, R. Mikolajczyk, M. Massari, S. Salmaso, G. S. Tomba, J. Wallinga, et al. Social contacts and mixing patterns relevant to the spread of infectious diseases. *PLoS Med*, 2008.

[221] A. N. Dajani, A. D. Lauger, P. E. Singer, D. Kifer, J. P. Reiter, A. Machanavajjhala, S. L. Garfinkel, S. A. Dahl, M. Graham, V. Karwa, H. Kim, P. Leclerc, I. M. Schmutte, W. N. Sexton, L. Villhuber, and J. M. Abowd. The modernization of statistical disclosure limitation at the u.s. census bureau. September 2017. [Online; posted September-2017].

[222] A. Narayan and A. Haeberlen. DJoin: Differentially private join queries over distributed databases. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[223] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2008.

[224] J. P. Near, D. Darais, C. Abuah, T. Stevens, P. Gaddamadugu, L. Wang, N. Somani, M. Zhang, N. Sharma, A. Shan, and D. Song. Duet: An expressive higher-order language and lineartype system for statically enforcing differential privacy. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2019.

[225] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[226] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[227] B. Nikolay, H. Salje, M. J. Hossain, A. D. Khan, H. M. Sazzad, M. Rahman, P. Daszak, U. Ströher, J. R. Pulliam, A. M. Kilpatrick, et al. Transmission of nipah virus—14 years of investigations in bangladesh. *New England Journal of Medicine*, 2019.

[228] A. Nilsson, P. Nikbakht Bideh, and J. Brorsson. A survey of published attacks on Intel SGX. Available from `https://portal.research.lu.se/portal/files/78016451/sgx_attacks.pdf`, 2020.

[229] H. Nissenbaum. Privacy as Contextual Integrity. *Washingtobn Law Review*, 79:119, 2004.

[230] K. Nissim, S. Raskhodnikova, and A. Smith. Smooth sensitivity and sampling in private data analysis. In *STOC*, 2007.

[231] K. Nissim, R. Smorodinsky, and M. Tennenholtz. Approximately optimal mechanism design via differential privacy. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, 2012.

[232] NYU and U. Austin. Pequin: An end-to-end toolchain for verifiable computation, snarks, and probabilistic proofs. `https://github.com/pepper-project/pequi`.

[233] Orchard codebase. `https://github.com/edoroth/orchard`.

[234] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 1999.

[235] A. Papadimitriou, A. Narayan, and A. Haeberlen. DStress: Efficient differentially private computations on distributed data. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2017.

[236] N. Papernot, M. Abadi, U. Erlingsson, I. Goodfellow, and K. Talwar. Semi-supervised knowledge transfer for deep learning from private training data. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[237] Y. J. Park, Y. J. Choe, O. Park, S. Y. Park, Y.-M. Kim, J. Kim, S. Kweon, Y. Woo, J. Gwack, S. S. Kim, et al. Contact tracing during coronavirus disease outbreak, south korea, 2020. *Emerging infectious diseases*, 2020.

[238] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

[239] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*. 2019.

[240] C. Peikert et al. A decade of lattice cryptography. *Foundations and Trends® in Theoretical Computer Science*, 10(4):283–424, 2016.

[241] J. C. Pickering and A. M. Fox. Enabling collaboration and communication across law enforcement jurisdictions: Data sharing in a multiagency partnership. *Criminal Justice Policy Review*, 2021.

[242] V. Pihur, A. Korolova, F. Liu, S. Sankuratripati, M. Yung, D. Huang, and R. Zeng. Differentially-private "draw and discard" machine learning. *ArXiv*, 2018.

[243] L. Pillaud-Vivien, A. Rudi, and F. Bach. Statistical optimality of stochastic gradient descent on hard learning problems through multiple passes. In *NeurIPS '18*, 2018.

[244] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis. The loopix anonymity system. In *Proceedings of the USENIX Security Symposium*, 2017.

[245] R. A. Popa, A. J. Blumberg, H. Balakrishnan, and F. H. Li. Privacy and accountability for location-based aggregate statistics. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[246] R. Pung, C. J. Chiew, B. E. Young, S. Chin, M. I. Chen, H. E. Clapham, A. R. Cook, S. Maurer-Stroh, M. P. Toh, C. Poh, et al. Investigation of three clusters of covid-19 in singapore: implications for surveillance and response measures. *The Lancet*, 2020.

[247] W. Qardaji, W. Yang, and N. Li. Understanding hierarchical methods for differentially private histograms. *Proceedings of the VLDB Endowment*, 6(14):1954–1965, Sept. 2013.

[248] D. Ramage and S. Mazzocchi. Federated analytics: Collaborative data science without data collection, May 2020. Google AI Blog, `https://ai.googleblog.com/2020/05/federated-analytics-collaborative-data.html`.

[249] S. Ramaswamy. Google "the keyword" blog: powering ads and analytics with machine learning. `https://blog.google/technology/ads/powering-ads-and-analytics-machine-learning/`, May 2017.

[250] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014.

[251] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *Proceedings of the ACM SIGMOD Conference*, 2010.

[252] J.-F. Raymond. Traffic analaysis: Protocols, attacks, design issues, and open problems. In *Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[253] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *SIAM Journal*, 8(2):300–304, 1960.

[254] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 157–168, New York, NY, USA, 2010. ACM.

[255] L. Reyzin, A. Smith, and S. Yakoubov. Turning HATE Into LOVE: Homomorphic Ad Hoc Threshold Encryption for Scalable MPC. Technical Report 997, 2018.

[256] P. Rindal and M. Rosulek. Faster malicious 2-party secure computation with {Online/Offline} dual execution. In *Proceedings of the USENIX Security Symposium*, 2016.

[257] R. L. Rivest, L. Adleman, M. L. Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[258] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[259] R. Rodrigues and P. Druschel. Peer-to-peer systems. *Communications of the ACM*, 53(10):72–82, oct 2010.

[260] S. Rogers and M. A. Girolami. A first course in machine learning. In *Chapman and Hall / CRC machine learning and pattern recognition series*, 2011.

[261] A. Roth and T. Roughgarden. Interactive privacy via the median mechanism. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2010.

[262] E. Roth, K. Newatia, Y. Ma, K. Zhong, S. Angel, and A. Haeberlen. Mycelium: Large-scale distributed graph queries with differential privacy. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[263] E. Roth, D. Noble, B. Hemenway Falk, and A. Haeberlen. Honeycrisp: Large-scale differentially private aggregation without a trusted core. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2019.

[264] E. Roth, H. Zhang, A. Haeberlen, and B. C. Pierce. Orchard: Differentially private analytics at scale. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[265] R. Rothblum. Homomorphic encryption: From private-key to public-key. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2011.

[266] A. Roy Chowdhury, C. Wang, X. He, A. Machanavajjhala, and S. Jha. Cryptϵ: Crypto-assisted differential privacy on untrusted servers. In *Proceedings of the ACM SIGMOD Conference*, 2020.

[267] T. Sander, A. Young, and M. Yung. Non-interactive cryptocomputing for nc/sup 1. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999.

[268] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[269] I. Security. X-force threat intelligence index 2022. `https://www.ibm.com/downloads/cas/ADLMYLAZ`, 2022.

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

[271] C. E. Shannon. Communication theory of secrecy systems. *The Bell system technical journal*, 28(4):656–715, 1949.

[272] E. Shi, T.-H. H. Chan, E. G. Rieffel, R. Chow, and D. X. Song. Privacy-preserving aggregation of time-series data. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[273] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. Membership inference attacks against machine learning models. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[274] D. L. Silverman. Developments in data security breach liability. *Bus. Law.*, 74:217, 2018.

[275] N. P. Smart and F. Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

[276] A. Smith. Differential privacy and the secrecy of the sample, Sept. 2009. `https://adamdsmith.wordpress.com/2009/09/02/sample-secrecy/`.

[277] K. Stokes and V. Torra. n-confusion: a generalization of k-anonymity. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pages 211–215, 2012.

[278] L. Story. A company promises the deepest data mining yet. *New York Times*, 2008.

[279] L. Sweeney. K-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

[280] M. Sweney. Google and viacom reach deal over youtube user data. *The Guardian*, 2008.

[281] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford. Scalable bias-resistant distributed randomness. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[282] J. Tang, A. Korolova, X. Bai, X. Wang, and X. Wang. Privacy loss in Apple's implementation of differential privacy on MacOS 10.12, 2017. `https://arxiv.org/pdf/1709.02753.pdf`.

[283] O. D. Thakkar, G. Andrew, and H. B. McMahan. Differentially private learning with adaptive clipping. 2021.

[284] C. Troncoso, M. Payer, J.-P. Hubaux, M. Salathé, J. Larus, E. Bugnion, W. Lueks, T. Stadler, A. Pyrgelis, D. Antonioli, et al. Decentralized privacy-preserving proximity tracing. *arXiv preprint arXiv:2005.12273*, 2020.

[285] S. Truex, N. Baracaldo, A. Anwar, T. Steinke, H. Ludwig, R. Zhang, and Y. Zhou. A hybrid approach to privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*.

[286] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[287] J. Van Den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[288] C. Veliz. *Privacy is Power: Why and How You Should Take Back Control of Your Data*. Penguin Random House, 2021.

[289] N. Volgushev, M. Schwarzkopf, B. Getchell, M. Varia, A. Lapets, and A. Bestavros. Conclave: Secure multi-party computation on big data. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2019.

[290] R. S. Wahby, I. Tzialla, abhi shelat, J. Thaler, and M. Walfish. Doubly-efficient zkSNARKs without trusted setup. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[291] X. Wang, A. J. Malozemoff, and J. Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[292] X. Wang, S. Ranellucci, and J. Katz. Global-scale secure multiparty computation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[293] Y. Wang, Z. Ding, G. Wang, D. Kifer, and D. Zhang. Proving differential privacy with shadow execution. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019.

[294] C. Weng, K. Yang, J. Katz, and X. Wang. Wolverine: fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[295] J. Whitehouse, A. Ramdas, R. Rogers, and Z. S. Wu. Fully adaptive composition in differential privacy. *arXiv preprint arXiv:2203.05481*, 2022.

[296] T. Wolverton. iPhone sales crater 15% in Apple's worst holiday results in a decade, and the forecast looks just as grim. *Business Insider*, Jan 2019. `https://www.businessinsider.com/ apple-q1-2019-earnings-iphone-sales-revenue-eps-analysis-2019-1`.

[297] D. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. Technical report, The Ethereum Foundation, June 2018.

[298] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: Scaling graph computation to the trillions. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2015.

[299] J. Xu, Z. Zhang, X. Xiao, Y. Yang, and G. Yu. Differentially private histogram publication. In *Proceedings of the IEEE International Conference on Data Engineering*, 2012.

[300] A. Yao. Protocols for secure computations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1982.

[301] D. Zhang and D. Kifer. LightDP: Towards automating differential privacy proofs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2017.

[302] H. Zhang, E. Roth, A. Haeberlen, B. C. Pierce, and A. Roth. Fuzzi: A three-level logic for differential privacy. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2019.

[303] H. Zhang, E. Roth, A. Haeberlen, B. C. Pierce, and A. Roth. Testing differential privacy with dual interpreters. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2020.

[304] J. Zhang, X. Xiao, and X. Xie. Privtree: A differentially private algorithm for hierarchical decompositions. In *Proceedings of the ACM SIGMOD Conference*, 2016.

[305] Y. Zhang, C. Wang, D. Pujol, J. Bater, M. Lentz, A. Machanavajjhala, K. Nayak, L. Vasudevan, and J. Yang. Poirot: private contact summary aggregation. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, 2020.

[306] M. Zhao, P. Aditya, A. Chen, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, B. Wishon, and M. Ponec. Peer-assisted content distribution in Akamai NetSession. In *13th ACM SIGCOMM Conference on Internet Measurement (IMC '13)*, Oct. 2013.

[307] E. Zheleva and L. Getoor. Preserving the privacy of sensitive relationships in graph data. In *Proceedings of the 1st ACM SIGKDD International Conference on Privacy, Security, and Trust in KDD*, 2008.

[308] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[309] B. Zhou and J. Pei. Preserving privacy in social networks against neighborhood attacks. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, 2008.

[310] W. Zhu, P. Kairouz, B. McMahan, H. Sun, and W. Li. Federated heavy hitters discovery with differential privacy. In *International Conference on Artificial Intelligence and Statistics*, 2020.

[311] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[312] S. Zuboff. *The Age of Surveillance Capitalism.* Profile Books, 2019.