Anonymous, Fault-Tolerant Distributed Queries for Smart Devices

EDWARD TREMEL, KEN BIRMAN, and ROBERT KLEINBERG*, Cornell University, USA
MÁRK JELASITY, Hungarian Academy of Sciences, Hungary and University of Szeged, Hungary

Applications that aggregate and query data from distributed embedded devices are of interest in many settings, such as smart buildings and cities, the smart power grid, and mobile health applications. However, such devices also pose serious privacy concerns due to the personal nature of the data being collected. In this paper, we present an algorithm for aggregating data in a distributed manner that keeps the data on the devices themselves, releasing only sums and other aggregates to centralized operators. We offer two privacy-preserving configurations of our solution, one limited to crash failures and supporting a basic kind of aggregation; the second supporting a wider range of queries and also tolerating Byzantine behavior by compromised nodes. The former is quite fast and scalable, the latter more robust against attack and capable of offering full differential privacy for an important class of queries, but it costs more and injects noise that makes the query results slightly inaccurate. Other configurations are also possible. At the core of our approach is a new kind of overlay network (a superimposed routing structure operated by the endpoint devices). This overlay is optimally robust and convergent, and our protocols use it both for aggregation and as a general-purpose infrastructure for peer-to-peer communications.

CCS Concepts: • **Security and privacy → Privacy-preserving protocols**; *Pseudonymity, anonymity and untraceability*; *Domain-specific security and privacy architectures*; • **Computer systems organization →** Sensor networks; Dependable and fault-tolerant systems and networks;

Additional Key Words and Phrases: Anonymous aggregation, data mining, overlay networks, smart meters

ACM Reference Format:
Edward Tremel, Ken Birman, Robert Kleinberg, and Márk Jelasity. 2018. Anonymous, Fault-Tolerant Distributed Queries for Smart Devices. *ACM Transactions on Cyber-Physical Systems* 1, 1, Article 1 (April 2018), 29 pages. https://doi.org/10.1145/3204411

1 INTRODUCTION

New distributed computing platforms are being created at a rapid pace as organizations become more data-driven, Internet connectivity becomes more widespread, and Internet of Things devices proliferate. For example, in proposals to make the electric power grid "smart," network-connected smart meters are deployed to track power use within the home and in larger buildings. The idea is that this data could be aggregated in real-time by the utility, which could then closely match power generation to demand, and perhaps even dynamically control demand over short periods of time by scheduling heating, air conditioning and ventilation systems. Such a capability could potentially enable greater use of renewable electric power generation and reduce waste.

*A portion of this work was completed while R. Kleinberg was at Microsoft Research New England.

Authors' addresses: E. Tremel, Ken Birman, and Robert Kleinberg, Computer Science Department, Cornell University; M. Jelasity, MTA-SZTE Research Group on Artificial Intelligence, University of Szeged, Hungary.

XXXX-XXXX/2018/4-ART1 $15.00
https://doi.org/10.1145/3204411

Yet the technical challenges of creating such a system are daunting, and they extend well beyond the obvious puzzles of scale, real-time responsiveness and fault-tolerance to also include public resistance to this form of universal monitoring infrastructure. Electric power consumption data can reveal a tremendous degree of detail about the personal habits of a homeowner, and customers are reluctant to share their smart meter data with the grid owner due to fears it will be used to profile them [38].

Today's most common approach is somewhat unsatisfactory: either the utility itself or some form of third party collects the sensitive data into large data warehouses, computes any needed queries against the resulting data set, and then shares the results with the higher level control algorithm. The data warehouse plays a key role, in protecting the consumers' privacy, but to do so, must be trusted and carefully protected.

The issue is not confined to smart grid uses; there are other types of cyber-physical systems in which the collection and storage of sensitive data is an obstacle to deployment. For example, a city or building owner might wish to query the images captured by a collection of security cameras to find criminal activity, but many people would object to collecting all the cameras' video feeds in a central location where they could also be used to track innocent citizens. The manager of a smart office building might query room-occupancy sensors to determine which parts of the building are inactive (and thus can have light and climate control turned off), but employees would not want this data to be used to identify who works the latest or comes in earliest.

Furthermore, all forms of data warehousing are under increased government scrutiny, particularly in Europe. Data has value: for example, the World Economic Forum has an activity that aims to create new legislative models for personal data protection [37]. Data is the "new oil" of our economy (as put by Meglena Kuneva, European Consumer Commissioner, in March 2009), and increasingly, is being treated as an asset that the customer owns and controls [21]. A data warehouse becomes problematic because it concentrates valuable information in a setting out of the direct control of the owner, and where an intrusion might cause enormous harm.

Here we describe a practical alternative: a decentralized virtual data warehouse, in which the smart devices collaborate to create the illusion of a data warehouse with the desired properties. The data can be queried rapidly through an interface that is reasonably expressive; while we don't support a full range of database query functionality, we definitely can support the kinds of queries needed for smart grid control or for other kinds of machine learning from smart devices. Focusing on the smart grid, our algorithm would allow power generation and demand balancing as often as every few seconds, which is more than adequate: in modern smart grid deployments load balancing occurs every 15 minutes, and even ambitious proposals don't anticipate region-wide scheduling at less than a 5-minute resolution.

Although computation occurs in a decentralized way, our algorithm ensures that individual smart meters have a light computational load centered on basic cryptographic operations involving small amounts of data and simple arithmetic tasks required for aggregation, such as computing sums. Similarly, the load imposed on any individual communication network link is modest. We assume a very simple and practical network connectivity model, and although we do require that the infrastructure owner (the power utility) play a number of roles, our protocol has a highly regular pattern of communication that can easily be monitored to detect oddities. We believe that this would be enough to incent the utility operator to behave correctly: the so-called *honest but curious* model.

The cryptography community has developed protocols that address some aspects of the problem we have described, notably homomorphic encryption and secure multi-party computation. Both methods use encryption and computation on ciphertexts to keep the values contributed by the

smart devices secret while they are being aggregated. However, neither approach is scalable or efficient enough for use in our target settings.

In contrast, our approach scales extremely well, is easy to implement (our experiments run the real code, in a very detailed emulated setting), and is quite fast. We offer several levels of privacy:

(1) In one configuration of our protocol, the smart meters trust one-another to operate correctly, and are trusted to not reveal intermediary data used in our computation to the (untrusted) system operator. Here, we can rapidly and fault-tolerantly compute aggregations. The system operator learns nothing except the aggregated result.
(2) A second configuration of our system is more powerful but a little more costly. Here we can support a much broader class of queries: we still focus on aggregation, but broaden our model to permit queries that prefilter the input data, and hence might include or exclude specific households. Further, in this second configuration, we assume that some bounded number of smart meters have been compromised and will behave as Byzantine adversaries. Nonetheless, we are able to fully protect the private data, by injecting noise in a novel decentralized manner. Here we achieve *differential privacy*.
(3) Beyond these two strongly private options, still further configurations of our protocol are also possible; of particular interest is one that could reveal a small random sample of anonymous raw data records to attackers, but (unlike differential privacy), gives exact query results.

In this paper we present our design for this new fault-tolerant, anonymous distributed data mining protocol, and prove its correctness. In a network of n nodes, query results can be computed in time $O(\log n)$ (the lower bound for a peer-to-peer network), and our anonymity mechanism introduces just a modest $O(\log n)$ inflation in the numbers and sizes of messages on the network. The solution can tolerate substantial levels of crash failures, can be configured to overcome bounded numbers of Byzantine failures, and is able to filter extreme data points while carrying out a wide range of aggregation computations. While our focus here is on aggregation, the novel network overlay protocol we introduce can also support a variety of other styles of peer-to-peer and gossip communication.

Specifically, the three main contributions of this paper are:

(1) A deterministic peer-to-peer overlay that is optimally efficient and fault-tolerant, and several protocols for anonymous and fault-tolerant message passing on this overlay.
(2) A decentralized anonymous query system based on this overlay network that can perform aggregate queries over client data without revealing anything about individual contributions.
(3) A design for a differentially private virtual data warehouse, based on the anonymous query system, that provides differentially private query results without a trusted third party and despite the presence of adversarial (Byzantine) client nodes.

The rest of this paper is organized as follows. First, in Section 2 we clarify the system model we are using and state our assumptions and goals. Section 3 discusses related work, including other approaches we rejected. Section 4 introduces our overlay network and lays out the details of our algorithm, Section 5 presents some extensions to it, and Section 6 provides proof of each version's correctness. Section 7 evaluates the practicality of our algorithm and presents some experimental results. In Section 8 we discuss how the peer-to-peer overlay we created as a part of this algorithm can be used to provide additional security in other peer-to-peer settings, and Section 9 concludes.

2 SYSTEM MODEL

Our target is a system set up and administered by a single owner or operator, with all participating devices connected to the same reliable network and logically within the same administrative

domain.[1] We assume that all client nodes (i.e. smart meters) are kept up-to-date on the list of valid peers (other clients) by a reliable membership service, run by a system administrator. We expect the set of clients to change fairly infrequently, since adding new smart meters or sensors to the network would require real-world construction effort, so the membership service should be trivial to implement. Furthermore, we assume that each client can be assigned an arbitrary integer ID by the system owner, and that the membership service also keeps nodes up-to-date on these IDs (which should only change when membership changes). Thus, in our system a node can choose any valid virtual ID and send a message to the node at that ID, and it knows the set of valid virtual IDs.

Many practical distributed systems [4, 19, 25, 36] assume that all nodes have well-known public keys and can digitally sign all of their messages. We also make this assumption; it requires just a standard PKI (public-key infrastructure), which the system owner can operate. Although the owner is not a client node, we also assume that the owner has a public key that is recognized by all the clients.

We consider the system owner or operator to be an honest-but-curious adversary, whose goal is to learn as much as possible about the clients (regardless of their consent) without disrupting the correct functioning of the system, and without wholesale compromise of the computing nodes. Thus, all plaintext messages sent on the network will be read by the owner, but the owner will not tamper with signed messages, decrypt encrypted ones, or attempt to impersonate smart meters. To prevent the owner from eavesdropping, the client nodes sign and encrypt all messages they send to other nodes using standard network-layer security (i.e. TLS [8]), and we will assume henceforth that all correct clients implement such encrypted communications. Note that this does not require us to assume that client devices are computationally powerful; TLS is an industry standard and even limited-resource systems often include hardware implementations of the needed functionality.

We consider two levels of trust for the client nodes (the smart metering devices). The fastest implementation of our protocol involves a level of trust: customers trust their own smart meters, but also those of other customers. These are assumed not to leak data to the system operator, and not to be compromised. A slightly slower version of our protocol makes much weaker assumptions, and can tolerate Byzantine (arbitrary and malicious) behavior by up to a bounded number of devices. Here, the devices might pretend to run our protocol, but secretly submit all measured data directly to the operator. Our solution for the former case yields exact answers to aggregation queries, and can tolerate fairly high levels of crash faults. For the latter case we inject noise that prevents the extraction of private information from the query results, hence we give inexact results, but have stronger protection guarantees (indeed, we can even support a wider class of queries and *still* offer stronger privacy guarantees).

Our system is built to tolerate client failures. We assume that up to t client nodes may fail during the process of executing a single distributed query. A crash failure includes any situation in which the client stops sending and receiving messages, whether due to loss of power, interruptions in network connectivity, or software failure on the client. When our protocol is configured to tolerate Byzantine failures, we bound the percentage of compromised nodes, but assume that Byzantine clients may deviate arbitrarily from the protocol. However, they cannot falsify the origins of the messages they send, and they cannot tamper with messages sent by other nodes, since honest nodes should only accept messages with valid digital signatures from the sender claimed in the message. Furthermore, since we assume that the devices have been registered with the operator, and trust the resulting list of devices, malicious clients are prevented from using Sybil attacks (as defined in [10]) to artificially increase the number of malicious nodes.

[1] This means that any participating device can communicate with any other device through the network infrastructure.

2.1 Goals

Now that we have defined the context of our system, we can give a more concrete definition of the goals of our data mining protocol. Assume that each client node starts with a single record, or data value, and that the system owner initiates a query by broadcasting a request that describes the desired aggregation operation, such as the computation of a histogram over the data values. By the end of the protocol, the system owner should receive a query result that is correctly computed from the values originating at correct, non-failed nodes.

We consider two categories of queries. Both compute aggregates: sums or other values that combine the input data, such as histograms. The first class of queries lacks any form of input filtering: if issued over a population, the output reflects the result of performing the requested aggregation over the full set of data values, counting each value exactly once. We refer to these as *unfiltered* aggregation queries. The second class of queries adds a pre-filtering step to the first class, for example selecting data only from certain households, or taking only data that satisfies some sort of predicate on the data items themselves. These *filtered* queries are more expressive, but create a risk that private data could be deanonymized, for example by running the same query twice, once including all households, and a second time excluding some particular target household: the delta is the data for that household. Accordingly, whereas we give exact results for unfiltered queries (the individual's data will be hidden in the aggregate), we offer the option of noise injection for the filtered case. Here, the gold standard is *differential privacy* [12], and we will see that our solution can achieve that guarantee, if desired.

We also consider two classes of client nodes. With trusted client nodes, only the query result should be released by any query. We are able to achieve this property for unfiltered queries. In contrast, if some of the client nodes might behave in a Byzantine manner, we need a stronger guarantee: because the Byzantine clients might publish any data they glimpse, we require that all data seen by client nodes must be anonymous and also contain injected noise. Further, we inject extra noise, so that the query result itself becomes imprecise. We can achieve differential privacy in this case, even with filtered queries.

Differential privacy turns out to come at a non-trivial cost. Accordingly, we also offer other configurations in which some anonymous data might be leaked by Byzantine nodes. Here, we do not achieve differential privacy, but we do gain performance and are able to give exact query results: a middle ground between what some might see as unwarranted trust (our first case), and what could be seen as excessive caution (the differential privacy case). This third option might be appealing in settings that demand very rapid queries, or where a noisy query result might not be acceptable.

3 RELATED WORK

Privacy concerns in the deployment of smart grids have been a popular topic for study since smart meters were introduced. Techniques from the area of Non-Intrusive Load Monitoring [22] can extract the time of use of individual electrical appliances from meter data, and Lisovich et al. [26] show that this information can be used to infer much about the personal habits of the home's occupants. Anderson and Fuloria point out in [3] that the current approach of most smart grid projects is to send all fine-grained smart meter data directly to a centralized database at the utility, where it can easily be accessed by employees and government regulators. McDaniel and McLaughlin [27] also survey the security and privacy concerns that can arise in smart grids.

The most widely known existing solutions to the problem of privacy-preserving data mining employ secure multiparty computation (MPC) or homomorphic encryption. In the former scheme, for each value of n (the size of the system) and each aggregation query, a special-purpose circuit is designed that combines values in ways that can embody a split-secret security mechanism and can

even overcome Byzantine faults. However, costs are high. For example, in [34] the authors remark that the size of an aggregating circuit will often be much larger than n, and that when this is the case, the overall complexity of the MPC scheme grows roughly as n^6. Furthermore, each step that a participant takes when executing a garbled circuit (i.e. evaluating a single gate) requires multiple cryptographic operations in advanced cryptosystems. Although this may be reasonable for clients that are desktop computers or servers, it represents a significant computational overhead for the embedded devices our system targets.

Homomorphic encryption schemes keep data encrypted with specialized cryptosystems that allow certain mathematical operations to be executed on the ciphertexts, producing a ciphertext that decrypts to the results of applying the same operations to the unencrypted data. This would allow clients to aggregate encrypted data, and decrypt only the query result, as in the system proposed by Li et al. [24]. However, fully homomorphic encryption (in which any function can be computed on ciphertexts) remains exponentially slow, and practical partially-homomorphic cryptosystems such as Paillier [32] allow only addition to be performed on encrypted data. Thus, systems based on partially-homomorphic encryption are restricted to only performing sum queries. Similar to MPC, aggregation using homomorphic encryption also requires each client to perform multiple expensive cryptographic operations on large ciphertexts, which represents a high computational overhead in our target environment of smart devices.

The problem we address is similar to those addressed by Differential Privacy, a framework first defined by Dwork [12, 13] which seeks to preserve the privacy of individual contributions to a database when computing functions on that database. In differentially private systems, a small amount of noise is added to the result of each query run on a database, to ensure that a curious adversary cannot analyze query results to determine any particular individual's contribution to the database. (In other words, the noise causes the margin of error of the result to be greater than a single individual contribution). The noise introduced is a Laplacian function proportional to the privacy-sensitivity of the query, which is a measure of how much a single record can affect its result.

However, Dwork's differential privacy model is normally formulated for a data warehousing situation, with a trusted third party. As noted, in our setting, there is no party that can be trusted to store the entire database. In fact we are not the first to explore extensions of differential privacy for use in a distributed setting. Two notable results are Dwork et al.'s distributed noise generation protocols [13], and Ács and Castelluccia's design for a smart metering system, in which each meter adds noise to its measurement locally before contributing it to the aggregate [2]. However, these systems still rely on MPC or homomorphic encryption to keep individual values hidden during aggregation, so they suffer the same high computational overheads. Our work innovates by achieving differential privacy in a very different way, at far lower costs and with much better scalability, as we discuss in Section 6.6.

In constructing our system, we will make use of a peer-to-peer overlay network for communicating amongst the clients. This network is based on gossip protocols, which were first developed by Demers et al. in [7]. We design our gossip-like protocol specifically to avoid interference by Byzantine nodes, a problem which has not seen nearly as much study as crash-fault-tolerant gossip. The most notable prior work on Byzantine-fault-tolerant gossip is BAR gossip [25]. The difference is that the authors assume that gossip is being used specifically to deliver a streaming broadcast from some origin node, and their solution is to use cryptographically "verifiable randomness" to prevent Byzantine nodes from picking gossip partners at will.

4 OUR ALGORITHM

Our algorithm is based on a carefully constructed overlay network that clearly defines how the client nodes communicate and makes it easy to extend the system to tolerate Byzantine faults. It is similar to a peer-to-peer gossip network, except it is completely deterministic instead of random. This allows nodes to independently determine when each phase of our protocol has finished.

Before describing our full data mining protocol, we will describe our overlay network and the ways nodes can communicate on it. Our approach assumes that the smart devices have direct peer-to-peer connectivity: for example, they might have built in wireless networking capability, similar to a smart phone, and the wireless partner of the utility could provide routing between devices. A "tunneling" model could also work: the devices could all connect back to the owner-operator, which would then provide routing at some central location. As will be shown below, our solution imposes only light networking and computing loads on the devices themselves. Although the aggregated traffic on a central routing system (for example in a tunneled implementation) might be moderately heavy, when one works out the numbers for even a fairly large regional deployment, they are well within the capabilities of modern network routers.

4.1 Building the Peer-to-Peer Overlay

Although our overlay network permits client nodes to communicate with each other in a peer-to-peer fashion, it does not use the fully decentralized model often seen in work on peer-to-peer systems, Traditional peer-to-peer systems assume that there is no way for a central server to manage membership in the network, so peer discovery becomes a hard problem that can be disrupted in many ways by Byzantine nodes (see, for example, [18]). As we described in Section 2, however, the data collection systems we target (such as the smart grid) already have a central server that monitors node connections to the network, which we will use to provide a basic membership service. Thus we can assume that every node in the network reliably knows the set of peers and their identities.

Our peer-to-peer communication system works as follows. Each client node is assigned a unique integer ID between 0 and n, where n is the total number of nodes in the system. Each client node also keeps track of the "round" of gossip it is in; like most gossip systems, we will describe it as if it takes place in synchronous rounds, but in practice it can be implemented asynchronously. We will define a function $g : \mathsf{ID} \times \mathsf{Round} \rightarrow \mathsf{ID} \times \mathsf{Round}$ that specifies the gossip partner for each node at each round of communication: if $g(i, j) = (a, b)$, then node i at round j should send a message to node a, which will receive the message in round b. We assume for now that the number of nodes n is a prime number such that 2 is a primitive root modulo n,[2] although we will revisit this assumption in Section 7.2. Under this assumption, we define the gossip function as

$$g(i, j) = ((i + 2^j) \bmod n,\ (j + 1) \bmod (n - 1)) \tag{1}$$

Note that the Round component of the function's output always refers to the round immediately following its input; for reasons which will soon become clear, we only number rounds 0 through $n - 1$ in this formal definition. The sequence of message propagation this function generates is shown in Figure 1a, from the perspective of node 0.

This function has two useful properties that allow the sequence of messages it prescribes to have the benefits of a random gossip system while being resilient to Byzantine behavior, namely *efficiency* and *uniformity*. In order to carefully define these properties, and prove that our function has them, we will need to make use of the following formalism, which represents each round of communications as a layer in a graph. For brevity, we will use the notation $[n] = \{0, 1, \ldots, n - 1\}$.

[2]This means that the sequence $2^1 \bmod n$, $2^2 \bmod n$, $\ldots$, $2^{n-1} \bmod n$ generates each integer between 1 and n exactly once. For example, 2 is a primitive root modulo 11 because $\{2^1 \bmod 11, \ldots, 2^{10} \bmod 11\} = \{2, 4, 8, 5, 10, 9, 7, 3, 6, 1\}$.

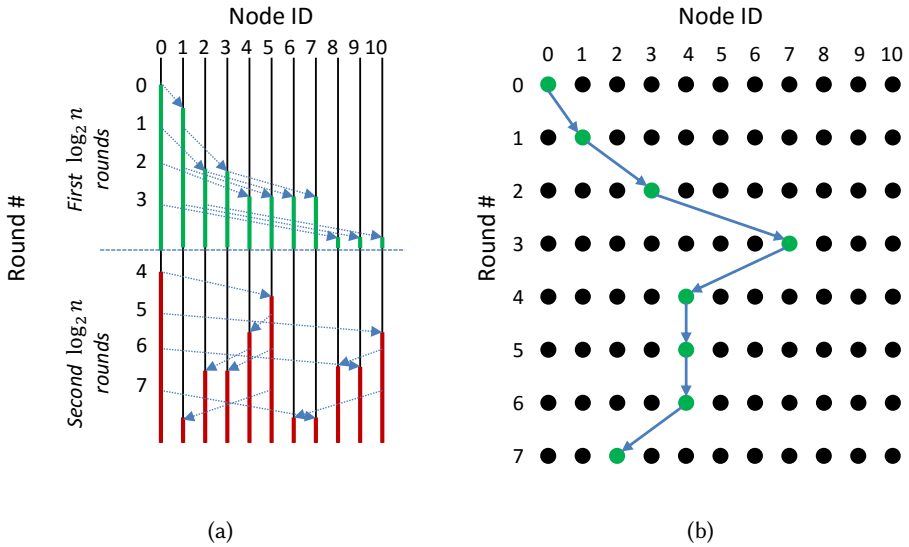


Fig. 1. (a) Information flow from node 0 in our scheme, showing two full epidemic cycles of $\log_2 n$ rounds each. Every process sends and receives one message per round; we omitted the extra messages to reduce clutter. Similarly, although each round can be viewed as a new epidemic, the figure just shows two.
(b) Example of the path taken by a tunneled message, from node 0 to node 3. This message would have 5 layers of onion encryption, since it has 5 different recipients.

Definition 4.1. A layered n-party gossip graph, denoted by $LG(n)$, is a directed graph with vertex set $[n] \times [n-1]$. The sets $[n] \times \{j\}$, for $j = 0, \ldots, n-2$, are called the layers of the graph. There is a permutation of the vertex set, denoted by g, which cyclically permutes the layers. In other words, for every vertex (i, j), $g(i, j)$ is a vertex of the form $(k, j+1)$ for some value of k (where the index $j+1$ is interpreted modulo $n-1$). The edge set of the graph contains exactly two outgoing edges from each (i, j): one of them points to $(i, j+1)$ and the other points to $g(i, j)$.

The edges in this graph represent the flow of information, which is why there is always an edge from (i, j) to $(i, j+1)$; it represents the fact that a message that reaches i in round j will still be in i's memory at round $j+1$. Now we can precisely define what it means for our function to be efficient:

Definition 4.2 (Efficient Gossip Property). A graph $LG(n)$ has the *efficient gossip property* if for all (i, j), the set of vertices reachable in $k = \lceil \log_2 n \rceil$ rounds is the entire layer $[n] \times \{j+k\}$.

Note that $\lceil \log_2 n \rceil$ is the fastest that data could possibly spread through the network in a peer-to-peer fashion if it starts at a single source. This property means our deterministic gossip function is as efficient at spreading information as the best-case scenario for traditional randomized gossip, and guarantees that any node can find a path to any other node in $\lceil \log_2 n \rceil$ hops while sending only messages approved by the function.

We can prove that the g we defined above has the efficiency property. Suppose we want to find a path from (i, j) to $(x, j+k)$. By choosing between the two outgoing edges at every level from j to $j+k-1$, we can find a path from (i, j) to $(i+y, j+k)$ whenever y is an integer representable as the sum of a subset of $\{2^j, 2^{j+1}, \ldots, 2^{j+k-1}\}$. Note that this is the same as saying that $y = (2^j)z$, where z is representable as the sum of a subset of $\{1, 2, 4, \ldots, 2^{k-1}\}$. Equivalently, $y = (2^j)z$ where z is any integer in the range $0, \ldots, 2^k - 1$. In particular, $y = x - i$ can be represented in this way by setting

$z = y/(2^j) \bmod n$. (Division by $2^j \bmod n$ is a well-defined operation, because n is an odd prime. Also note that $y/(2^j) \bmod n$ belongs to the set $\{0, ..., n-1\}$ which is a subset of $\{0, ..., 2^{k-1}\}$.)

The other useful property of our peer-to-peer communication network is its uniformity, which we define as follows:

Definition 4.3 (Uniformity Property). A graph $LG(n)$ has the *uniformity property* if for all distinct a, b in $[n]$, there is exactly one value of j such that $g(a, j) = (b, j + 1)$.

In a graph with this property, each node gossips with every other node exactly once before gossiping with the same node again. This minimizes the risk that crash failures cause a split in the gossip network, since all possible paths are used equally often. It also makes it difficult for Byzantine nodes to target their malicious behavior at a particular victim, since they will get only one chance in n to send a legitimate message to that victim, and honest nodes can discard messages that do not come from the sender prescribed by the function.

It is straightforward to prove that the g we defined above has the uniformity property. Note that the equation $g(a, j) = (b, j + 1)$ translates to $2^j \bmod n = b - a$ when using our definition of g, and there is a unique j satisfying this equation by our assumption that 2 is a primitive root modulo n.

4.2 Communicating on the Overlay

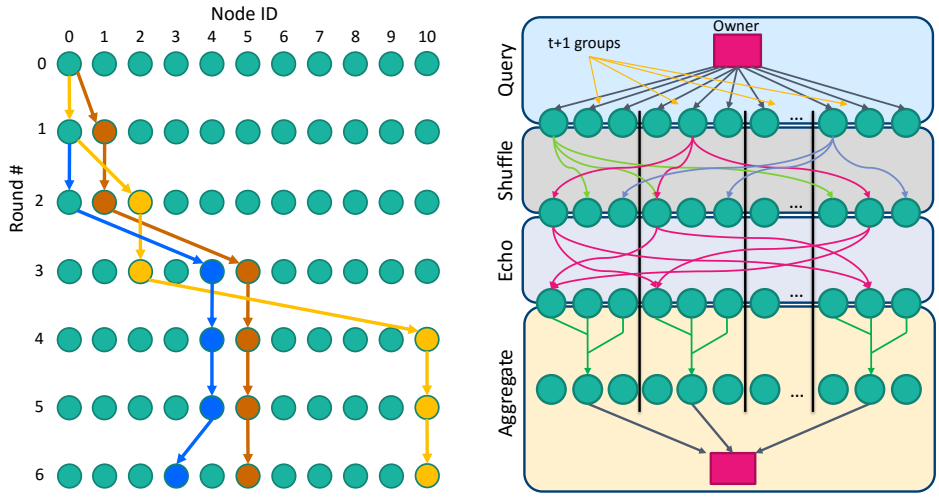
There are three ways in which we use our overlay network to send messages between nodes: "tunneled" or "onion-routed" message sending, disjoint multicast, and flooding. The first two can be combined for a tunneled multicast, which uses onion routing for each path in the multicast. Each of these communication methods are used in different stages of our data aggregation protocol.

To send a tunneled message through the network, sending node a finds a path to receiving node b by performing a breadth-first search on the graph generated by function g for the current round. A "path" is any sequence of transitions through the graph (including remaining at the same node for a round) that includes at least $\lceil \log_2 n \rceil / 2$ nodes; we place this minimum length restriction in order to preserve the sending node's anonymity, as explained in Section 6.5. For example, Figure 1b shows one possible path from node 0 to node 3 through a network of 11 nodes. Such a path will always exist within $2\lceil \log_2 n \rceil$ rounds of the current round, since the graph is efficient (it would be $\lceil \log_2 n \rceil$ if we did not specify a minimum length). Then a encrypts its message with an encryption onion, where each layer corresponds to one node along the path and can only be decrypted by the private key of that node (similar to onion routing [33]). At each layer of encryption, the node includes an instruction for the node that can decrypt that layer indicating how many rounds it should wait before forwarding the message.

By relaying messages in this manner, a can be assured that no intermediate nodes will learn the message it is sending to b, or the fact that a is the sender and b the recipient of a message. Each node in the path will learn only its immediate predecessor and successor when it relays the onion, and assuming messages are continuously being sent around the network, even the first node in the path will not know it is the first node because a could have relayed the message from a previous sender.

While nodes are relaying tunneled messages, they may receive more than one encrypted onion that they need to forward to the same successor, or receive an encrypted onion along with a normal (unencrypted) message. To minimize time and bandwidth overhead, all these messages should be forwarded at once in the same signed "meta-message."[3]

[3]Practically, this can be implemented by sending the messages in sequence over a TLS connection.



(a) (b)

Fig. 2. (a) A disjoint multicast from node 0 to nodes 3, 5, and 10. (b) Overview of the design of the basic crash-tolerant aggregation protocol. In the Shuffle and Echo phases, only a subset of the message paths are shown, and arrows of the same color carry the same tuple.

To send a disjoint multicast to a set of k nodes, a needs to find a set of k node-disjoint paths through the overlay network to the receivers.[4] These paths can be found by a repeated breadth-first-search of the graph from a to the receivers, in which the nodes on the path to a receiver are deleted once the receiver is found. Our experiments suggest that these paths will have length at most $k + \lceil \log_2 n \rceil$, as long as the total number of elements in the paths ($k\lceil \log_2 n \rceil$) is of the same order of magnitude as $\sqrt{n}$. For values of n above roughly 10,000, this property holds, and our system targets large sizes in this range. Node a then adds to each of k copies of the message instructions for each intermediate node, indicating how many rounds it should wait before forwarding the message in order to follow the path a has chosen, and sends them out. The node-disjoint property of the paths means that they are failure-independent, which minimizes the impact of node failures on this multicast: t failed nodes can prevent at most t recipients from receiving the message. Figure 2a illustrates a node performing a disjoint multicast to three recipients. Note that even with n as small as 11, $k = 3$ disjoint paths can still be found in $k + \lceil \log_2 n \rceil = 7$ rounds.

The combination of these two communications primitives is the tunneled multicast. In a tunneled multicast, node a finds a set of k node-disjoint paths to its chosen k recipients, as in the regular multicast, but then encrypts each copy of the message with an encryption onion and puts the instructions for each intermediate node in that node's encryption layer, as in tunneled messaging.

If the sending node wishes to prevent nodes other than the desired recipients from reading the message, but does not care about remaining anonymous, it can simply encrypt each copy of the message with the recipient's public key before sending it along the path to that recipient. The forwarding instructions are kept in the clear, so the intermediate nodes do not need to do any work decrypting an onion layer. We will refer to this variant as an encrypted multicast.

Finally, a node can send a message by flooding in a manner very similar to a standard epidemic gossip algorithm. To flood a message, node a adds a time-to-live field to the message initialized to

[4]To clarify, these are paths that do not have any nodes in common.

$\lceil \log_2 n \rceil + t$, where t is the number of failures tolerated, and begins sending it to its gossip partners. Upon receipt of a flood message, a node should decrement the TTL field, and forward the message to its next gossip partner if the TTL is still positive. Flooded messages are guaranteed (by the efficiency and uniformity properties) to reach every node in the network in $\lceil \log_2 n \rceil + f$ rounds, where f is the actual number of failures, so nodes can stop forwarding a flood message when its TTL reaches 0. Flooding can be used for broadcasts, or it can be used to send a message to a single recipient in a highly fault-tolerant manner by encrypting the body of the message with the recipient's public key.

To simplify failure detection and speed up our protocol, we require that every node in the network send a message on every round that the overlay is running. If a node has no messages to send or forward in that round, it should simply send an explicitly empty message. This way nodes do not need to wait for the entire message timeout interval to conclude that their predecessor is not sending a message.

Discussion. Even if operated in a network smaller than the target size mentioned above, or with an unusually high rate of failures, our solutions will not fail catastrophically. The primary risk is of a gradual degradation of guarantees. For example, if a multicast sender can't find enough node-disjoint paths along which to relay data, the algorithm could consider paths that share just a single node (followed by paths that share two nodes, and so on). Thus we could still run our protocol, but now some nodes would find themselves in more than one path.

The failure of such a node would disrupt more than one path, and with enough such failures, data from a healthy node might not be properly relayed. But notice that this will depend on a very specific set of nodes failing, and only in certain rounds, and because nodes have no control over their position in the overlay, the weakness would be very hard to exploit.

In future work we plan to explore this question experimentally, but we believe that our solution would remain quite useful even in such cases. With a higher probability of disrupted rounds, the risk is a gradually increasing possibility of message delivery failure. For the aggregation protocol given below, this would manifest as queries that fail to produce a result and must be resubmitted, or that undetectably omit some inputs. In a smart grid or a similar setting, where the input data itself is of limited quality in any case, such outcomes might well be acceptable, particularly if the infrastructure owner knows that they are highly unlikely.

4.3 Basic Crash-Tolerant Aggregation

Using the overlay network we have defined above, we can build an algorithm for fault-tolerant, anonymous data mining. Assume for the moment that the query is unfiltered and that client nodes are trusted: they may fail by crashing, but will not disclose data to the system operator and are permitted to glimpse data contributed by other client nodes. We will still seek a basic privacy guarantee, namely that any data originating on some other node is seen only in an anonymous form, and that no client system can see more than a very small number of these randomly selected anonymous records.

At a high level, responding to a query with this algorithm involves three phases: Shuffle, Echo, and Aggregate. In the Shuffle phase, nodes send their values to a set of proxies who will contribute the values to the query on their behalf, using onion routing to hide the source of the values. In the Echo phase, proxies for the same value echo their values to each other to accommodate failures during shuffle. Finally, in the Aggregate phase, each subset of nodes that contains a single proxy for each value conducts binary-tree peer-to-peer aggregation; this phase is the only one that does not use the overlay. The phases of the algorithm are sketched in Figure 2b.

Shuffle Phase. First, the owner broadcasts its desired query function to all nodes, using ordinary direct messages on the network. Upon receiving a query, each node picks $t+1$ proxy nodes, choosing one at random from each sequence of $\frac{n}{t+1}$ consecutive node IDs. Each sequence of $\frac{n}{t+1}$ IDs forms an *aggregation group*.[5] Then each node forms a tuple $(R, v, [p_1, p_2, \ldots, p_{t+1}])$ where R is the query number (a monotonically increasing value set by the owner when it broadcasts the query), v is the value it will contribute to the query, and $[p_1, p_2, \ldots, p_{t+1}]$ is the list of proxies. It uses a tunneled multicast to send this tuple to its chosen proxies, along $t + 1$ disjoint paths through the overlay. After $t + 1 + 2\lceil \log_2 n \rceil$ rounds of communication, all messages should have reached their destination. If a node receives a tuple with a query number higher than the one it is currently processing, it buffers the tuple and waits to receive a query request broadcast from the owner (assuming that the owner's message has been delayed longer than the peer-to-peer message). If it receives a tuple from a past query (a lower query number), it should discard that tuple.

Echo Phase. At the end of the Shuffle phase, each proxy will have approximately $t + 1$ tuples containing a value and a list of other proxies. Each tuple indicates a proxy group that the node belongs to, where a proxy group is simply the set of nodes that are all proxies for the same value. Within each proxy group, though, not all nodes may actually have the tuple due to failures along the path from the origin node. To resolve this problem, each node encrypted-multicasts a copy of each proxy value it holds to the t other nodes that should proxy the same value. (Onion encryption is not necessary, since the values are now being sent from a proxy, not the node that contributed them.) After another $t + 1 + 2\lceil \log_2 n \rceil$ rounds of communication, all echo messages should have reached their destination.

Aggregate Phase. This is the only phase in which we do not use our overlay network for node-to-node communication. Instead, nodes within each aggregation group (as defined in the setup phase) communicate directly with each other. Although it would be possible for us to conduct aggregation within the groups using only the overlay network, this would re-introduce failures into groups that contain only healthy nodes (since communications must be routed through the entire network), and increase the number of messages that must be sent.

Within each aggregation group, nodes use a binary tree to aggregate the values into a leader, along with a count of the number of participating nodes. A tree can be induced on the group by a simple ordering of the node IDs, making the lowest half of the IDs the leaves, and adding rows in ascending order. A non-leaf node should wait for an incoming message with the current query result, then combine its values with the intermediate result, increment the participation count by the number of values it contributed, and send the new query value to its parent. In order for tree aggregation to produce correct results, query functions must be associative and commutative. While this does preclude some types of queries, it is not as restrictive as addition-only queries (which is the restriction for systems based on homomorphic encryption).

Finally, all $t + 1$ leaders send their results to the system owner, along with the count of how many nodes participated. If their values differ, the owner should accept the result that has the most contributions. In order to accommodate the case where a leader node has failed, the owner should wait for a fixed timeout after receiving each result instead of waiting for $t + 1$ results.

This achieves our first goal: we now have a solution in which a trusted set of client systems collaboratively compute the result of an unfiltered aggregation query. Although the client nodes do glimpse a small randomized subset of anonymous records, this is not a sufficient amount of data to enable any client node to reconstruct the entire data set, or to attempt a de-anonymization attack on the system.

[5]Nodes could be divided into aggregation groups by any deterministic function; we use consecutive ID sequences because it is the simplest.

5 EXTENSIONS TO OUR ALGORITHM

The basic version of our protocol presented in Section 4 is intended for a system with a small number of crash failures. It does not work as well when the number of failures is larger than $O(\log n)$, and it is not designed to tolerate Byzantine failures. In this section we will describe some alternate versions of our algorithm that can handle these cases. The Byzantine fault-tolerance method injects noise, and because this noise introduces very strong protection, that version of our protocol can be used even with filtered queries.

5.1 Tolerating High Failure Rates

Sending messages along independent paths through the overlay to avoid faults minimizes the number of nodes that must relay each message (reducing communications overhead), but this method becomes inefficient when t is larger than $O(\log n)$ due to the difficulty of finding so many independent paths. Instead, if the number of failures is expected to be a large percentage of n, the Shuffle and Echo phases can be replaced with two phases of flooding. In the Scatter phase, nodes flood $t + 1$ encrypted copies of their values to randomly chosen relay nodes, and in the Gather phase, the relay nodes flood their encrypted values to the proxy nodes that can decrypt them.

Scatter Phase. First, the owner broadcasts its desired query function to all nodes, and each node picks a proxy from each of $t + 1$ aggregation groups, as in the base protocol. In addition, for each proxy that a sending node has chosen, the sender picks a "relay" node for that proxy, uniformly at random from the set of all nodes not chosen as proxies. The sender creates a copy of its tuple for each proxy, encrypted with the public key of that proxy. It then floods these encrypted tuples to their relay nodes, using the single-recipient version of flood in which the message is encrypted with the recipient's public key. (This means each tuple is inside a simple two-layer onion, with the outer layer encrypted for the relay, and the inner layer encrypted for the proxy). After $\lceil \log_2 n \rceil + t$ rounds of flooding, every message will have reached every node, and nodes can discard messages they cannot decrypt.

Gather Phase. Once the Scatter phase has finished, each relay node begins flooding all of the tuples it received and decrypted (each relay node will have approximately $t + 1$). Since these tuples are already encrypted with the public keys of the proxy nodes that should receive them, this is equivalent to a single-recipient flood, but the relay nodes do not need to do any additional encryption. After another $\lceil \log_2 n \rceil + t$ rounds of flooding, every message will have reached every node, which means all healthy proxy nodes have received all of their proxy values.

Aggregate Phase. Unchanged from the base protocol.

5.2 Tolerating Byzantine Failures

In some cases, such as when the system is at risk of viruses infecting some of the client nodes, it might be necessary to tolerate Byzantine failures rather than crash failures. We have also developed a Byzantine-fault-tolerant version of our protocol, which adds an additional setup phase and a Byzantine agreement phase in order to protect against malicious nodes. As noted, here we can support filtered queries as well as unfiltered ones.

Setup Phase. First, the owner broadcasts its desired query function to all nodes, as in the base protocol. Upon receiving a query, each node still chooses one proxy uniformly at random from each aggregation group, but there are now $2t + 1$ aggregation groups (and they are defined as sequences of $\frac{n}{2t+1}$ consecutive IDs). Each node forms a tuple $(R, v, [p_1, p_2, \ldots, p_{2t+1}])$ as before, except that we potentially add a noise factor to the data; the noise injection is discussed in detail in subsection 6.6. It blinds the tuple by combining it with a random secret value, asks the owner to sign the ciphertext,

then unblinds the signed tuple, which produces a cleartext tuple signed by the owner. (This is the blinded signature scheme first described by Chaum in [6]).

Shuffle Phase. This proceeds as in the base protocol, except it takes $2t + 1 + 2\lceil \log_2 n \rceil$ rounds of communication to complete. If a node receives a tuple that does not have a valid signature from the system owner, it should discard that tuple.

Byzantine Agreement Phase. This replaces the Echo phase from the base protocol. It is still the case that some nodes within a proxy group for a tuple may not actually have the tuple, since Byzantine nodes along the path from the origin node may have refused to relay the message, or Byzantine origin nodes may have sent a tuple to only some of their proxies. To resolve this problem, nodes within a proxy group conduct two rounds of multicasts among themselves.

First, each node in a proxy group that has actually received a tuple signs the tuple with its private key and encrypted-multicasts it to the other nodes. Once all messages have been received, each node counts the number of copies of the tuple it has that are signed with distinct valid signatures. If a node receives t or fewer distinct signatures for the tuple (including its own), it rejects that value and does not act as a proxy for it. If a node receives at least $t + 1$ distinct signatures it concatenates them all into a single message, signs it, and encrypted-multicasts this message to all the other nodes. A node that receives such a message accepts it if it contains at least t signatures that are different from the message's signature, and adds the tuples contained to its set of signed tuple copies. Then each node decides to use a value if and only if it has received at least $t + 1$ distinct signatures for it, and deletes the extra copies of the tuple. This is essentially the two-phase Crusader agreement algorithm described by Dolev in [9], with the simplification that Byzantine nodes cannot change the values they are multicasting (because they are signed by the owner), so the decision is only on the presence or absence of a single possible value.

Aggregate Phase. This proceeds as in the base protocol, with the exception that the count of participating nodes is not needed, because the owner can simply use the query result that it receives at least $t + 1$ times. Since at least $t + 1$ out of $2t + 1$ aggregation groups contain only correct nodes, the owner should receive $t + 1$ identical query results from the leaders of those groups, so a result that appears at least $t + 1$ times is correct.

6 PROOFS OF CORRECTNESS

For each version of our algorithm, we will demonstrate that it satisfies all the goals we defined in Section 2.1. First, we will show that the system owner always receives an aggregation result that includes all values contributed by honest nodes. For the versions that tolerate only non-Byzantine faults, this consists of showing that the owner will receive the correct query result despite t failures.

6.1 Basic Version

In the basic version of our protocol, we start by proving that values from non-failed sources will always reach their non-failed proxies. In the Shuffle phase, since each source node sends its values along $t + 1$ node-disjoint paths, at most t of those paths can contain a failed node (no node appears in more than one path). This means at least one proxy in each proxy group must have received its value by the end of the Shuffle phase, since the path from the source to that proxy contained no failed nodes. Then, in the Echo phase, each node in a proxy group sends its value to the other nodes along $t + 1$ node-disjoint paths. Due to the uniformity property of our overlay graph (gossip partners repeat only once every n rounds), these are different paths from the paths used in the Shuffle phase, and hence they will not be affected by the same failures. Since there are only t total failures, failures can only occur in these paths if they did not occur in the Shuffle phase. Thus, either a value reaches its proxy during the Shuffle phase, or it reaches it by a different path in the

Echo phase. This means that every proxy that has not itself failed will have the source's value by the end of the Echo phase.

Now we show that there is at least one correct aggregation group. With $t + 1$ aggregation groups, at least one aggregation group is guaranteed to contain no failed nodes. Since every source node must have chosen a proxy in that group, and all source values must have reached their non-failed proxies, that group contains one copy of every value contributed by a source node. The query result that the failure-free group returns to the owner will have the maximum possible count of contributions, so it will always be accepted as the correct answer by the owner.

6.2 High-Failure Version

The high-fault-tolerant version's correctness stems from the efficiency property of our overlay, which guarantees that a message from any node can be flooded to all nodes in $\lceil \log_2 n \rceil$ rounds in the absence of failures. Since the number of nodes "infected" with a flooded message doubles each round, a single failure can delay the convergence of the flood by at most one round; even if the failure occurs at the beginning of the flood (the worst case), the sending node will send the message to a non-failed node in the next round, and effectively start a new flood with a different tree of nodes one round later. This means that when a source node starts flooding its encrypted tuple to relay nodes in the Scatter phase, the encrypted tuple is guaranteed to have reached every non-failed node after $\lceil \log_2 n \rceil + t$ rounds. Similarly, when a relay node starts flooding an encrypted tuple to proxy nodes in the Gather phase, the message is guaranteed to reach every non-failed node in $\lceil \log_2 n \rceil + t$ rounds. This means that every encrypted tuple will have reached every non-failed relay by the end of the Scatter phase, and all non-failed proxy nodes that had non-failed relay nodes are guaranteed to have received their tuples by the end of the Gather phase.

It may seem like some cause for concern that both the relay and the proxy for a value need to be non-failed in order for that value to reach its proxy. However, note that the relay nodes are chosen to be distinct from the proxies. This means that each failure can either mean that a proxy failed, or that a relay node failed, but not both. Thus each failure of a relay or proxy prevents exactly one proxy from having a sender's value (either because the proxy itself fails or because the proxy's relay fails). Essentially, a relay failure is equivalent to a proxy failure, and there are at most t of either kind. Since the t failures can prevent at most t proxies from learning the sender's value, each sender is guaranteed to have its value reach at least one non-failed proxy.

The proof that there is at least one correct aggregation group is the same as with the base protocol. There must be one aggregation group that contains no failed nodes, and every source has a proxy in that group that is non-failed. This group will return the correct answer to the owner.

6.3 Byzantine-Fault-Tolerant Version

Although Byzantine nodes can exhibit arbitrary behavior, we have constructed our system such that most actions that do not fit within our protocol will have no effect on the system. For example, Byzantine nodes cannot successfully impersonate correct nodes (even if they try) because correct nodes will not accept a connection without a valid digital signature, and they know the public keys of all the other nodes. They cannot contribute more than one value (each) to a query, because in the Setup phase the owner will only sign one (blinded) value from each node per query, and correct nodes will not use a value with the wrong query number. Also, they cannot send messages to nodes that are not their prescribed gossip targets, because every node can independently compute the overlay network function and will reject messages that should not have been sent in the current round. Since the membership server is reliable, the malicious nodes also cannot use a Sybil attack [10] to artificially increase the number of malicious nodes.

As a result, there are only two kinds of malicious behavior that we need to be concerned with in proving that the algorithm runs correctly: stopping messages from propagating by dropping them, and sending a message to some nodes while withholding it from others. The first behavior is covered by our tolerance of crash failures, while the second is nullified by the Byzantine Agreement phase.

Note that in the Shuffle phase, at least $t + 1$ proxies are guaranteed to receive the value sent by a source node, for the same reasons that at least one proxy is guaranteed to receive the value in the basic protocol.

In the Byzantine Agreement phase, each proxy group can have at most t Byzantine nodes in it, so it has at least $t + 1$ correct nodes. At the beginning of this phase, some correct nodes may not have the proxy value due to Byzantine nodes along the path to a correct node. Furthermore, if the source of the value was a Byzantine node, it may be the case that only Byzantine nodes have the proxy value, because the source Byzantine node refused to send it to correct nodes.[6] However, the fact that correct nodes only accept values that have been signed by $t + 1$ distinct nodes guarantees that any value accepted by a correct node will also be accepted by every other correct node.

In order for a value to get $t + 1$ signatures, it must have arrived at $t + 1$ different nodes, and there are only t Byzantine nodes. If a value is accepted in the first multicast step of this phase, this means it must have been seen by at least one correct node. Therefore at least one correct node will send the $t + 1$ signatures to every other node in the second multicast step, and any correct nodes that receive this message will also accept the value. Since honest nodes choose disjoint paths to their destinations for multicasts, Byzantine nodes will only be able to prevent signatures from reaching honest nodes if they are not in the proxy group – the disjoint paths can include group members only as endpoints. Each Byzantine node that blocks a multicast message to one recipient guarantees one additional honest node in the proxy group, which means one additional unique signature will be sent to all recipients. Thus no Byzantine node can reduce the number of distinct signatures received by an honest node by more than 1, so a value seen by an honest node can always achieve $t + 1$ signatures at all honest nodes.

Restricting communication in the aggregation phase to only the processes within an aggregation group has a similar benefit to restricting communication to the overlay network in previous phases: it limits the nodes that Byzantine participants can effectively communicate with. The aggregation groups for the aggregation phase are deterministically defined and public knowledge, so if a Byzantine node attempts to send messages to nodes outside its group, those nodes can discard them as easily as they can discard out-of-order gossip messages. This means that at most t aggregation groups contain any Byzantine nodes, and at least $t + 1$ are composed only of correct nodes.

The correct-only groups all start with the same set of values, since by the end of the Byzantine Agreement phase all correct proxies within each proxy group have received and accepted the same value. Thus *the $t + 1$ correct-only groups will all compute the same result.* Regardless of what the t aggregation groups with Byzantine nodes in them compute, the owner will receive $t + 1$ identical results from the correct groups, and will accept their value as the answer.

6.4 Preventing Data Pollution

For the BFT version of our protocol, it is not sufficient to prove that the owner receives a query response that includes all contributed values. We must also consider the possibility that the Byzantine nodes contribute wildly incorrect data in order to make the query result inaccurate, even though it completes successfully. Such pollution attacks can have a substantial impact; for example, in one highly visible event during 2008, Amazon S3 directed all writes to a single server for a period

[6]Conversely, if the source of the value was not Byzantine, then at least one correct proxy has the value, which is how we know that any value contributed by an honest node will be included in the aggregate.

of nearly 8 hours. The issue was ultimately blamed on a faulty aggregation participant that kept asserting that the server in question had an infinite amount of free space [1]. In what ways does our protocol protect against this attack?

Our first line of defense is to ensure that any single node can only contribute a single value towards the query. As we mentioned above, this is guaranteed by the owner's signature in the Setup phase. Thus a faulty node can only impact the aggregate by providing a value that is extreme in some sense.

To protect against extreme outliers, we employ a second line of defense. As we proved in the previous section, the aggregation value ultimately used by the owner is computed entirely by correct nodes in subgroups that have only correct participants, and all of those subgroups employ the same sets of values. Any Byzantine value is thus included by all, or excluded by all.

This setup makes it easy to eliminate bad data by including a conditional clause with each query that will only include reasonable values. Since queries can be any function that is associative and commutative, a conditional clause that selects only values that fit some statically-defined criteria could easily be included. For example, based on historical records, a power utility might know that no matter how extreme the weather, individual household power use will always be in the range of 0 kW/h to perhaps 2.5 kW/h. It could thus submit a query that selects only values in this range. If a Byzantine node were to then claim a consumption of -5 kW or 1.2 GW for a four hour period, that value would be filtered out by honest nodes when they apply the query function in the Aggregate phase. The Byzantine nodes cannot avoid this filtering, since it will be applied by all honest nodes computing the query, and at least $t + 1$ aggregation groups contain only honest nodes.

6.5 Privacy

Now we will show that our algorithm meets our privacy goals of preventing both the system operator as well as individual client nodes from learning more than a small number of anonymous individual records. First, it is impossible for any node to learn the identity of the node that contributed a particular value during the aggregation process. In the basic and BFT versions of our protocol, the Shuffle phase effectively anonymizes the senders of the values, in the same way that onion routing anonymizes the senders of Internet packets, by hiding values inside encrypted containers that reveal only one step of the routing path at a time. Intermediate nodes do not learn the value they are forwarding, and destination nodes have no way of tracing back the path that led to them. The minimum length of the onion-routed paths also ensures that destination nodes cannot guess at the sender of a value based on how quickly it arrived, since all messages will take a minimum number of rounds to arrive.[7] Every node sends and receives a message in every step of the protocol, hence traffic analysis would not be fruitful. In the high-failure-rate version of our protocol, the combination of the Scatter and Gather phases anonymizes the senders, since relay nodes cannot see the values they are relaying, and by the time any proxy node receives a value, enough overlay rounds have elapsed that (by the efficiency property of our overlay network) the value could have originated at any node.

In the non-Byzantine versions of our protocol, nodes will not share data with each other by deviating from the protocol, and during the Aggregate phase, the nodes only send intermediate query values, not individual records. Therefore, each node only learns individual anonymous values when it receives them as proxy values in the Shuffle or Gather phases. Each node is chosen as a proxy by $t + 1$ different origin nodes on average, so in expectation, each node learns $t + 1$ anonymous values. If these nodes do not communicate to the owner, the owner itself does not learn

[7] Specifically, since all paths are required to be at least $\lceil \log_2 n \rceil / 2$ nodes long, and our overlay guarantees that any node is reachable from any other node in at most $\lceil \log_2 n \rceil$ hops, any of $n/2$ nodes could be a possible origin for a message that arrives in the minimum time.

any individual values from these versions of the protocol, because all communications between nodes are encrypted and only the query result is sent back to the owner. With respect to the client nodes themselves, an honest but curious client node will glimpse anonymous records, but only a small number of them, at the protocol step where blinded data is extracted back into raw form. As a result, over time, a client node can build up a statistical picture of the overall database. However, this kind of picture could have been explicitly computed using an aggregation query that randomly samples data, hence it reveals nothing that was not already available in the system.

In the BFT version, we must assume that Byzantine nodes may share information with each other with out-of-protocol messages. Thus, although the number of individual records learned by any one node is still limited to the number of senders that chose it as a proxy ($2t + 1$), the t Byzantine nodes could combine their records to see an expected total of $2t^2 + t$ anonymous values. Since for the BFT version of the protocol (as with the base version) we expect t to be bounded by $\lceil \log_2 n \rceil$, the number of records learned by a Byzantine node is at most $O(\log^2 n)$.

Without noise injected, this clearly represents a leak, and with sufficient auxiliary data, the leak could compromise privacy. For example, suppose that an attempt is being made to spy on a particular home, and the home happens to have two electric hot water heaters that can both be scheduled to respond to signals from the smart grid. This could be sufficiently distinctive that any raw data record that reports a count of two such units is very likely from the target home, and will very likely be revealed to the intruder. This motivates the stronger option we now explore, in which our protocol becomes more complex, but can be fully secured against such attacks.

6.6 Differential Privacy

In the smart metering scenario, the most common form of data aggregation involves a summation over some set of real numbers: for example, power consumption over a given short period, or anticipated power need, or the amount of power that can be scheduled (consumed early, or late, if the utility needs a bit of help shaping demand to match the available generation capacity). We might have thousands of consumers, and each successive aggregation query will typically reflect different data, since the underlying power needs of the household vary fairly rapidly.

As described earlier, the fundamental anonymity protection in our algorithm is through a data shuffle phase that anonymizes private (raw) data. When the aggregated local data is a single number, this data shuffle suffices because—even if Byzantine nodes share the raw data samples they observe—re-identification of anonymous samples that are glimpsed just occasionally would, in general, not be feasible.

However, we might want to support the aggregation of more complex data, such as high-dimensional vectors. If a single household contributes a long vector of data, each element of which is somehow specific to that home, it becomes much more likely that the individual record could embody patterns that would enable compromised nodes to de-anonymize the random sample to which they gain access: in effect, a more detailed form of raw data might sometimes be attacked using forms of auxiliary information that could actually be available. We might also want to support filtered queries, where certain households are included or excluded based on ID or based on some properties. Such an extension would be easy to implement: the query for a given iteration of the protocol is already disseminated by the owner/operator at the start of that iteration, and could certainly include a filtering action. Thus all that would be required is to have the smart devices execute the desired preliminary filtering before contributing their data. Unfortunately, this would trivially allow the utility to learn the raw data of any given household in the present setup, by submitting a filtered query that excludes all but one household.

In this section we sketch our solution to achieve differential privacy that in turn would make it possible to extend our application scenarios and support high-dimensional data and filtered queries

with privacy protection. The definition of differential privacy was given in [11]. For some single iteration of querying, let M be an algorithm producing an answer to a query issued on any possible database $D \in \mathcal{D}$, where $\mathcal{D}$ represents the possible databases that can be created by prefiltering data in various ways at the start of the round. Then, when computing a query on D, algorithm M should also introduce random noise, thereby randomizing its output. That is, for a fixed database D, $M(D)$ will be a random variable. Let the distance function $d : \mathcal{D} \times \mathcal{D} \mapsto \mathbb{N}$ be defined as the number of records in which two given databases differ. Without loss of generality, we assume that all the databases contain the same number of records (for example, if the customer's data is excluded by the filter, a node could simply contribute default data).

Definition 6.1. (*ε-differential privacy*) Let M be a randomized mechanism acting on databases. M is *ε-differentially private* iff for any two fixed databases D and D' such that $d(D, D') = 1$, and for any output M, we have

$$P(D|M) \leq P(D'|M) \cdot \exp(\varepsilon). \tag{2}$$

We've expressed the traditional definition in a Bayesian style. This definition is equivalent to the usual definition [11] if the prior distribution over the databases is uniform (any possible database is equally likely, that is, $P(D) = P(D')$).

We focus on the sum query from now on. One possible way to achieve differential privacy in a database D_i is for M to compute a a noise value calibrated according to the *sensitivity* of the query, and then add this value to the query result [14]. For example, in the case of the sum query, we can return

$$M = M(D) = Y + \sum_{i=1}^{n} v_i, \tag{3}$$

where Y is an appropriate random variable. A common choice for the distribution of Y when v is one-dimensional is $Y \sim \text{Laplace}(0, Z/\varepsilon)$, where Z is a constant representing the *global sensitivity* of the query function [12, 14]:

Definition 6.2. (*Global sensitivity [14]*) The *global sensitivity* Z_f of $f : \mathcal{D} \mapsto \mathbb{R}$ is given by

$$Z_f = \max_{D, D' :\ d(D,D')=1} |f(D) - f(D')| \tag{4}$$

This approach can be generalized to higher dimensional data using appropriate vector norms to determine sensitivity and to define the high-dimensional noise vector Y [14].

Turning to our scenario, the obvious possibility is to simply inject noise in the aggregation output, thereby applying the standard machinery of differential privacy but performing the action in a decentralized manner, as in [13] or [2]. These approaches involve having each smart device inject noise into its raw data before ever contributing it within the system, and selecting this noise in a purely local manner, in such a way that the final aggregated noise will follow the desired distribution, as a function of some fixed privacy parameters. In a setting where the infrastructure is trusted and the operator only sees the query results, such a solution would suffice.

In our setting, however, Byzantine nodes are able to glimpse a random subset of anonymous but unencrypted data records, and must be assumed to share them with the query operator. Thus a level of noise adequate to protect the query results would not necessarily be sufficient under the same ϵ-differential privacy requirement: this level of noise does not protect individual data records. The proxying step, in effect, leaks private information.

One option would be to try to use differential privacy techniques to protect the individual data contributions themselves, but this proves unsatisfactory. Suppose that each raw data item has a sufficient level of noise injected at the outset so that even relaying proxy nodes observe only differentially private information. In this setup, each item must be considered a query that returns

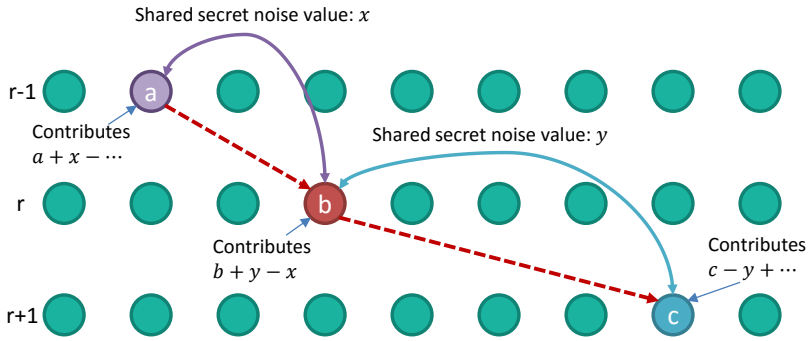


Fig. 3. Example of the paired noise generation scheme. If a query starts on round r when node b has predecessor a and successor c, it will add a noise value that it shares with c and subtract a noise value that it shares with a before contributing its data to the query.

a single record, which has a sensitivity equal to the value itself (calculated in the same way as that of the sum query). Hence each node is required to inject noise of the same order of magnitude as the value it intends to contribute. Naively summing such values results in an overall noise with a variance of size $O(n)$, which is so high as to make the query result useless.

Fortunately, the regular structure of our overlay network offers a simple remedy. Suppose that we pair nodes in the following manner: in overlay round r, each node a communicates to some node b. If we consider the pairing (a, b), we obtain a set of node-pairs that changes with each round. Each node will appear in two such pairs: once as the sender, and once as the receiver.

Next, notice that because we assume a PKI, the Diffie-Hellman protocol enables us to construct a shared secret for each such pair, without requiring further communication: each node knows the public key of the other as well as its own private key, and this suffices to parameterize the Diffie-Hellman method without further exchange of information. It follows that every matched pair (a, b) knows the matching for round r and also has access to a shared secret that can be used when this matching arises. The same pairing will repeat every n rounds, and we do not wish to reuse identical random noise values, so we use the shared secret as the seed for a random sequence.

Accordingly, when a run of the query protocol starts, any node i can determine its two counterparts for the current round of the overlay: one to which i sends, and one from which i receives. We can use the shared secrets to select two pseudo-random noise values, such that both members of each pair select the same noise value for that same query. The distribution of this noise value must be tuned to provide differential privacy for an individual record, as described above. In a pair (a, b), the "sender" node, a, should add this random noise to the raw data contributed by a for this query. Simultaneously, the receiver, b, should subtract that same noise value but from the raw data that b will contribute. (Figure 3 illustrates the association between node pairs and noise values). Since each node is a member of two pairs, each node thus modifies its raw data by adding one noise value, and subtracting another, and no other node has access to both secrets. However, when we aggregate these values to compute the sum, all or most of the noise (depending on failures) will cancel out.

Accordingly, we obtain our desired solution by combining the two methods. First, the paired-noise scheme is used to hide the raw data from our Byzantine nodes. Next, we also have our aggregation nodes add additional (unpaired) noise to the aggregated output data value, proportional to the sensitivity of the overall query. Now the paired noise cancels, but the additional noise is still

included into the aggregate. This additional noise enables us to assert that the query result will be differentially private.

With this change, any anonymous random sample constructed by the compromised Byzantine nodes contains heavily noised values. Lacking any way to know the level of noise that was injected, or any way to match the pairs of values, forwarding nodes have no possibility of de-noising the raw data. This is true even though the matching is public, and even if the nodes sending to and receiving from some particular node i were both Byzantine and reveal the amount of noise that they injected: a compromised proxy node would know that there exists a record that originated with some particular smart meter containing this specific level of noise, but would have no basis for identifying that record during the forwarding step. Thus, our scheme does not leak any private data through actions or data available to compromised nodes.

Crash failures complicate the picture by allowing the aggregated noise to depart from the analytic goals: although compromised nodes cannot force the exclusion of data, if non-compromised nodes fail by crashing, we might now include noised inputs that won't have compensating negative noise inputs. For example, if node a crashes during a query that starts on round r, then there will be two uncompensated noise contributions: one (a positive contribution) from the node that sent to a in round r, and one (negative) from the node to which a was scheduled to send in round r. However, notice that because these noise values were independently drawn from the same distribution, and one was added but the other subtracted, their expected sum is 0. Moreover, the variance of the uncompensated noise can be shown to be proportional to the node failure probability, which is expected to be low in most applications. If the infrastructure operator dynamically adapts the overlay to eject faulty nodes (and later to readmit them once they are repaired or connectivity is reestablished), the duration of such effects could also be limited. Thus we believe the issue would not be an obstacle to practical use of our technique, and it does not give compromised nodes any opportunity for disruptive behavior beyond the potential that already was present.

7 EVALUATION

To evaluate how well our protocol works, we will theoretically analyze some of its performance properties, then describe our implementation and experimental tests.

7.1 Overheads

All versions of our protocol introduce some amount of communication overhead in order to tolerate faults. However, our protocol is still efficient, scaling only with the logarithm of the size of the network. For the basic version, there are $2t + 2\lceil \log_2 n \rceil + 2$ rounds of communication on the overlay network: the Shuffle phase takes $t + \lceil \log_2 n \rceil + 1$ rounds of communication, since a tunneled multicast to k nodes takes $k + \lceil \log_2 n \rceil$ rounds and $k = t + 1$, and the Echo phase takes another $t + \lceil \log_2 n \rceil + 1$ rounds for a disjoint multicast. The Aggregate phase can be considered one additional round of communication per node, since each node needs to send only one message in this phase, so there are $2t + 2\lceil \log_2 n \rceil + 3$ rounds of communication. Since t is at most $\lceil \log_2 n \rceil$ for the basic version of the protocol, this means the protocol will finish in at most $4\lceil \log_2 n \rceil + 3$ rounds of communication, which is $O(\log n)$. We believe this to be practical, and also to represent a tight lower bound: one cannot aggregate data from n sources in fewer than $O(\log n)$ steps using a peer-to-peer protocol.

The BFT version also scales logarithmically. There are $6t + 3\lceil \log_2 n \rceil + 3$ rounds of communication on the overlay network: the Shuffle phase is one tunneled multicast, and the Byzantine Agreement phase is two disjoint multicasts, but now $k = 2t + 1$. Each node completes a round of communication with the owner in the Setup phase, and one additional round in the Aggregate phase, so there are $6t + 3\lceil \log_2 n \rceil + 5$ messages. Since t is at most $\lceil \log_2 n \rceil$ for the BFT protocol, this means the protocol will finish in at most $9\lceil \log_2 n \rceil + 7$ rounds, which is $O(\log n)$.

The high-failure-rate version of the protocol has a higher communication overhead in order to accommodate more than $\lceil \log_2 n \rceil$ failures. There are $2t + 2\lceil \log_2 n \rceil$ rounds of communication on the overlay network ($t + \lceil \log_2 n \rceil$ for each of the first two phases), plus one round for the Aggregate phase, so the protocol takes a total of of $2t + 2\lceil \log_2 n \rceil + 1$ rounds. However, this value cannot be bounded by $O(\log n)$ since $t > \lceil \log_2 n \rceil$. Also, note that the number of messages sent per round in this protocol is much larger than the number of messages per round sent in the other versions. In the other versions, each node will be used in on average $t + 1$ or $2t + 1$ different disjoint paths, so on any one overlay round a node will need to send at most $t + 1$ or $2t + 1$ messages (each containing an encrypted proxy value). In this version, since all nodes flood their encrypted tuples through the entire network, on a given overlay round a node may need to forward up to $n \cdot t$ messages.

Our protocol also has low computational overhead. The only cryptography involved is digital signatures, public-key encryption, and the symmetric encryption used in TLS connections. In all versions, for most of the protocol each node only needs to compute a single public-key encryption per round (to set up a TLS connection with its target). The Shuffle and Byzantine Agreement phases are the most expensive. In the shuffle phase each sending node must compute $O(\log n)$ public-key encryptions per destination node in order to construct the encrypted onions for paths whose lengths range from $\log_2 n/2$ to $\lceil \log_2 n \rceil$, which is $O(\log^2 n)$ total encryptions per node. The BFT version of the protocol has an additional expense in the Byzantine Agreement phase. First, each node needs to sign each proxy value it has received; since each node will receive on average $2t + 1$ proxy values, this is $O(\log n)$ signatures per node. Then each node must compute $O(\log n)$ public-key encryptions per value that it multicasts, in order to encrypt the message with the target's public key. Since each node participates in on average $2t + 1$ proxy groups, and there are two multicasts per group, this is $O(\log^2 n)$ total encryptions per node (in addition to the encryptions performed in the Shuffle phase).

7.2 System Size

We should take a moment to note that our assumption in Section 4.1 that the network size n was a prime with primitive root 2 is practical. Artin's primitive root conjecture [29] asserts that, asymptotically, the density of primes having primitive root 2 converges to a constant, known as Artin's constant, whose value is roughly 0.374. The conjecture is true assuming the Generalized Riemann Hypothesis, as was shown by Hooley [15] in 1967. We verified experimentally that suitable values of n are sufficiently dense, which makes it easy to find one that is very close to the actual network size.

Specifically, we computed all the suitable sizes up to 20,000,000, and found 475,333 of them, which is indeed about 37% of the prime numbers in the same range as predicted by theory. Figure 4 shows the histogram of the gaps between consecutive suitable sizes. Similarly to the distribution of gaps between primes, this distribution is approximately exponential. More importantly, we can observe that any actual size can be approximated with a very high precision, and the relative precision actually increases with size. The unused node IDs that result from "rounding up" n to the nearest suitable prime can either be doubly assigned (giving a few nodes a second ID) or treated as failed nodes if there are much fewer than t of them.

7.3 Experiments

Our protocol is designed for networks of embedded devices such as smart meters, and we did not have access to sufficient numbers of such devices to test the protocol on real hardware. However, we created a detailed software simulation of the smart grid in which to test our algorithm. Our simulation uses a probabilistic model of electricity consumption, based on the one developed by Paatero and Lund in [31], to generate a realistic electrical load at each of n simulated homes. Each

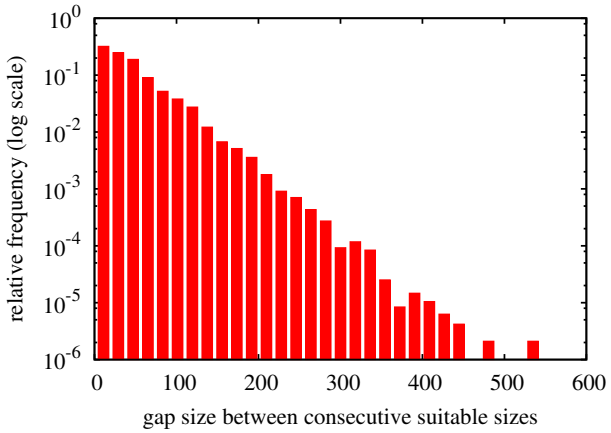


Fig. 4. Histogram of the distances between a suitable prime and the next suitable prime, based on network sizes up to 20,000,000.

home has an associated "meter" that continuously records the electricity being consumed and can communicate with any other meter by sending a message through a simulated utility network. The simulated network chooses latency delays between 2 and 10 ms for each message (using a normal distribution with a mean of 4, plus a fixed constant of 2), which we believe represents the latency to be expected from a dedicated network connecting meters in a small local area. The simulation is event-driven to model an asynchronous system in which each meter reacts independently to the event of receiving a network message. Failures are modeled by choosing a fixed set of meters to fail at the beginning of a query, and preventing them from sending or receiving messages for the duration of the query. For the variants that expect only crash failures (not Byzantine failures), non-failed meters can detect that a meter has failed when they attempt to send a message to it, since a non-malicious meter that has crashed will also fail to respond to TCP connection requests.

Paatero and Lund's model does not include home heating or cooling appliances, but these represent the largest source of electrical load in most households and present the best opportunity for demand-side load management via programmable thermostats. Therefore, we added central air conditioners, window air conditioners, and furnace fans to the model as possible devices that could generate load at a home. We used data from the American Housing Survey [35] for the penetration rate (frequency of occurrence) of air conditioners and furnaces, and information from several online datasheets (e.g. [5] and [23]) for the per-cycle energy consumption of these devices.

We implemented the basic version of our protocol in this simulation and ran an experiment in which the utility sent a query to the meters every half-hour asking for two sum values: the total energy consumption in kilowatt-hours since the previous query, and the total energy consumed by demand-side manageable devices since the previous query (representing load that is available for shifting). Figure 5 shows the data that the simulated utility collected from our system over 24 hours, when running with 1019 simulated meters, and compares the query results with the true sum of energy consumption recorded at meters locally (obtained by inspecting the global simulation state).

In addition, we used our simulation to measure the time and bandwidth costs associated with running our protocol. First, we measured the amount of time it took for a single query to complete as the size of the system scaled up, using the simulator's internal clock. In order to more accurately model the time it would take to complete a query, we simulated the overhead of cryptography computations at each node in addition to the delays caused by network latency, using benchmarks

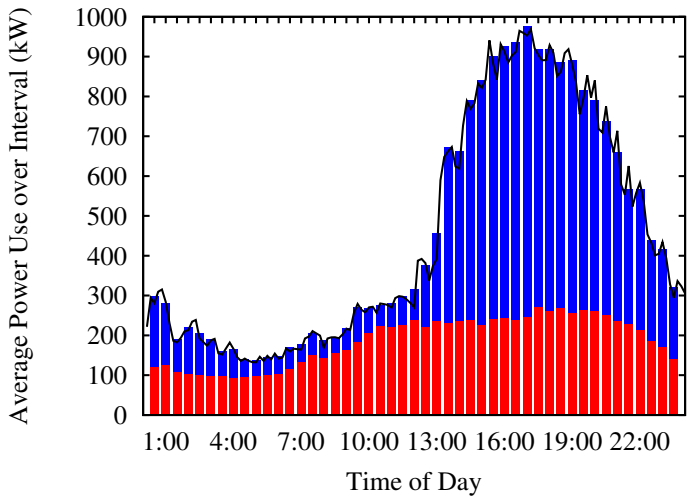


Fig. 5. Data collected by the utility, running queries using our system. Red bars are reported non-shiftable load, while blue bars are reported load from devices with DSM potential (i.e. heating and cooling systems). The black line is the actual consumption recorded by the meters locally.

measured on OpenSSL to determine the amount of time each RSA encryption, decryption, and signature would take. Figure 6 shows the delay experienced by the system owner between broadcasting a query and receiving a reliable result from the nodes, in simulated milliseconds, for the basic and BFT variants of our protocol. For each system size n, t is set to $\lceil \log_2 n \rceil$, the largest feasible value for these variants, and we ran the experiment once with no failures and once with t failures. This experiment shows that queries can be completed in a few seconds even for large networks of devices. Failures, and their associated timeouts, impact the running time of the query much more than an increase in the system size.

Second, we measured the approximate amount of data that each node would need to send over the network as the size of the system scaled up. We did this by assuming that each encrypted value-tuple object had a size of 1kb, and counting the number of such messages that each meter sent during the execution of a single query. Figure 7 shows the average total data sent per smart meter during a query execution for the basic and BFT protocols, again using $t = \lceil \log_2 n \rceil$. Note that failures do not significantly affect the amount of data sent; in fact, they slightly decrease it, since in our model failed nodes do not contribute any tuples to the aggregate (they fail at the beginning of the query). As the figure shows, even with a large system and the redundancy necessary for Byzantine fault tolerance, each node needs to send only a few megabytes of data. The amount of data sent scales with t, and since $t = \lceil \log_2 n \rceil$ it increases only logarithmically.

We also implemented the highly crash tolerant version of our protocol in our simulation, and ran similar experiments on it, with $t = 0.1n$. These experiments are still in progress, but preliminary results show that it is similar to the BFT protocol in terms of speed: with 797 meters, a query completes in 1.3 s with no failures, and 8.1 s with 10% failures.

Finally, to test our overlay's usefulness for general peer-to-peer applications, we implemented an asynchronous version of the overlay network in PeerSim [28], and used it to run a basic epidemic gossip algorithm, in which a source node tries to spread its value to all the nodes in the network. We then measured the number of rounds of communication it took for a value to propagate to all nodes with different levels of random node failures, and compared this convergence rate to

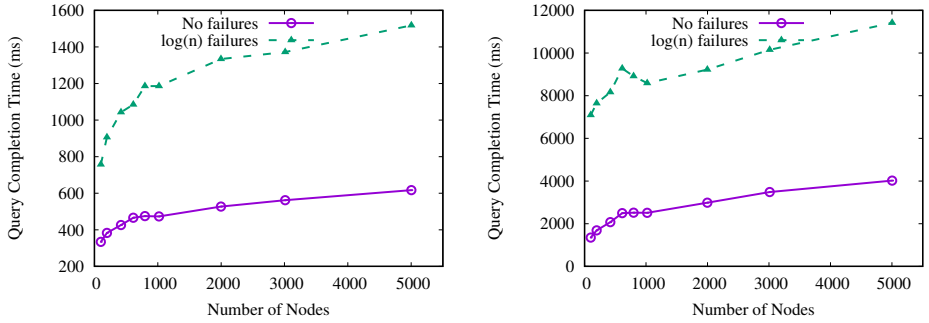


Fig. 6. Time for a single query to complete, in simulated milliseconds, for the basic protocol (left) and the BFT variant (right)

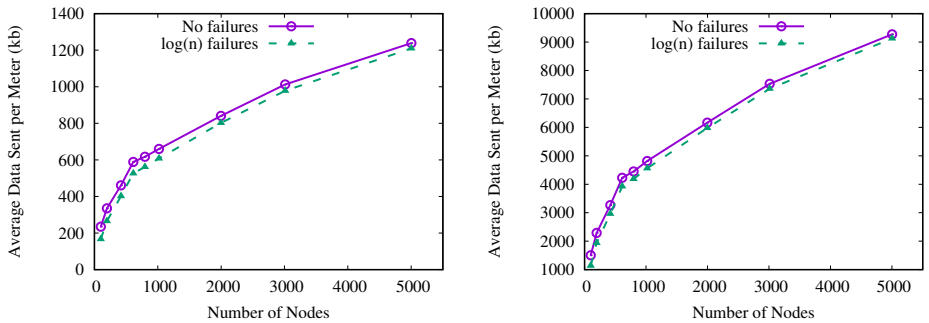


Fig. 7. Average data sent per smart meter for a single query, assuming each simulated message is 1kb, for the basic protocol (left) and the BFT variant (right)

an epidemic of the same size using a standard random gossip protocol. Figure 8 shows that node failures delay convergence by only a few rounds, as predicted by the properties of our overlay. In fact, even with 50% node failures our overlay converges reasonably quickly, and significantly faster than random gossip. This is because our overlay graph has a high degree of connectedness, which provides multiple redundant paths by which data can reach each node and results in many different opportunities for a node to learn about data that it previously missed due to a failure.

8 OTHER USES OF THE OVERLAY

Besides its uses in our aggregation protocol, the deterministic peer-to-peer overlay network we described in Section 4.1 is very general-purpose. It provides many of the same features as a gossip system, while providing better tolerance of both crash and Byzantine failures. In fact, it is more efficient at broadcasting messages through a network than random gossip, because its derandomized partner selection mechanism means that nodes always gossip with the optimal peer for spreading data to "uninfected" nodes.

As our tests in Section 7.3 showed, our overlay is highly resilient to crash failures. In addition, it is much more resistant to Byzantine behavior than random peer-to-peer communications. In a random gossip system, Byzantine nodes can send bogus gossip messages at a rapid rate to any or all of the other nodes, which must accept and process them since honest nodes have no way of determining whether their peers' choices are truly random. This allows malicious nodes to target a

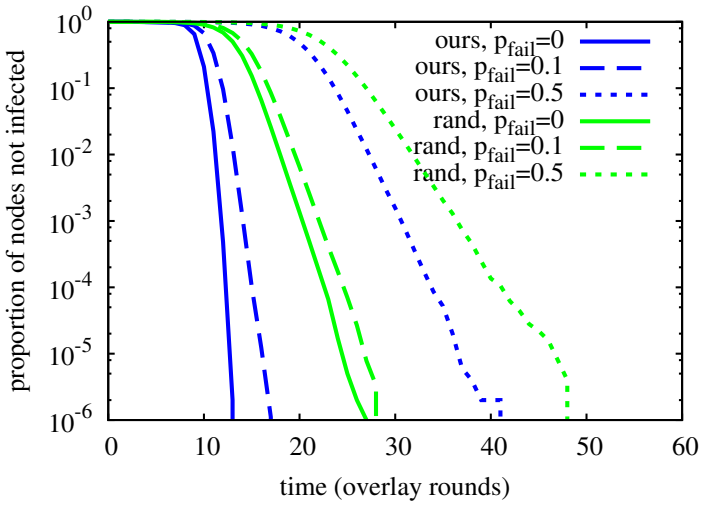


Fig. 8. Comparison of our overlay with random push gossip with several node failure probabilities.

single victim node for a denial of service attack by flooding it with messages, or introduce as much invalid data as they want with no limitations on the number of times they contribute relative to honest nodes. By contrast, all nodes in our system can independently and efficiently compute the peer-selection function, so every node knows exactly which node it should be receiving a message from in a given communications round. Honest nodes can thus quickly reject too-frequent or out-of-order messages, and Byzantine nodes are limited to participating only within the framework of the correctly functioning overlay.

There are many existing distributed systems applications that use peer-to-peer communications to spread data around a network, which could benefit from being adapted to use our overlay. For example, Astrolabe [36] is a lightweight data storage system that uses gossip to keep records up to date, Fireflies [19] is a network monitoring and intrusion detection system that uses regular peer-to-peer exchanges to monitor node health, and T-Man [17] is an overlay topology management system that itself uses gossip to set up other overlays. Our overlay would not only make these systems more robust, it would make them faster, because it guarantees that their gossip phases complete in exactly $\lceil \log_2 n \rceil$ rounds.

On a more basic level, several peer-to-peer distributed computation algorithms have been proposed that are gossip-like but spread computations instead of data around a network. These include gossip-based aggregation [20], which is a simple distributed aggregation scheme, distributed peer-to-peer learning [30], which performs stochastic gradient descent based on random walks through an overlay network, and chaotic matrix iterations [16], which builds a machine learning model by repeatedly passing it between members of a peer-to-peer network. Such algorithms could be made more Byzantine-fault-tolerant by using our derandomized peer-to-peer network instead of random peer selection. In addition, they can be made more accurate, because our network removes the possibility of sampling bias that would be caused by random gossip choosing to include some nodes multiple times and skip others completely.

9 CONCLUSION

In this paper we have addressed the problem of securely and privately computing aggregate queries over data stored in a distributed system. The approach is unusual in that we completely eliminate

the need for a trusted third party data warehouse. Instead, we treat the collection of devices as a virtual data warehouse, and they execute the query through a distributed, collaborative protocol. The solution is scalable, quite robust, and very inexpensive. As such, we believe it offers a practical alternative for systems with clients that are too weak to use expensive cryptography, and yet where it is important to minimize disclosure of private data.

Compared to existing data-aggregation schemes that rely on cryptography, the primary tradeoff of our system is that it requires additional communication overhead in exchange for lower computational demands. Homomorphic-encryption-based aggregation schemes send the encrypted data directly from the clients to the data analyst, while our system requires clients to relay their data through several other nodes before sending it to the analyst in order to provide privacy. In a setting where network bandwidth is readily available but processing power is not, we believe our system is the appropriate choice.

We actually offer three variants on our protocol. The first is optimized for speed, and can tolerate crash failures; it is provably private so long as the smart meters can be trusted, and if the class of queries doesn't include any that filter input data by deciding node by node whether or not to include data for particular devices. A second much more general version of our protocol can support filtered queries and is also Byzantine fault-tolerant. Moreover, this version achieves differential privacy for sums and other aggregates where fractional noise can be injected into the raw data in such a way that it will aggregate to achieve a target noise level corresponding to an appropriate noise distribution. However, the extra protection costs us performance, and the results of the queries contain noise (noise injection is required in the differential privacy model). Other configurations are also possible; most interesting of these is one that might slowly leak a randomized anonymous sample from the underlying raw data, but gives exact answers to unfiltered queries even with Byzantine attackers.

We envision our methods being used in systems operated by an honest but curious entity such as an electric power distributor, who carries out a computation that employs aggregated data or other information collected at large scale to perform a desirable function, such as optimizing power delivery. The user would choose the lowest-overhead version of our system that meets their privacy and fault-tolerance needs. For example, utility companies will have regulatory and corporate-policy requirements for safeguarding user privacy (which dictate whether differential privacy is necessary), and may also have dependability and accuracy requirements (which dictate the level of fault-tolerance necessary); they will choose the most efficient algorithm that can meet these requirements.

In large systems such as the smart grid where data privacy is a concern, the lack of a practical and scalable method for aggregating data while preserving privacy has prevented the implementation of many useful data mining features. We hope that our solution will enable progress, and that it might also prove useful in other kinds of large-scale systems where it is necessary to aggregate and analyze private data safely.

ACKNOWLEDGMENTS

Our work was supported, in part, by grants from the US National Science Foundation and DARPA. R. Kleinberg was partially supported by NSF grant CCF-1535952. This research was supported by the Hungarian Government and the European Regional Development Fund under the grant number GINOP-2.3.2-15-2016-00037 ("Internet of Living Things"). We are also grateful to Ari Juels for a suggestion that simplified our differentially private noise injection protocol, and to Al Demers for many helpful conversations in developing our overlay communications protocols.

REFERENCES

[1] 2008. Amazon S3 Availability Event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html. (July 2008). Accessed: 27 Jan 2015.

[2] Gergely Ács and Claude Castelluccia. 2011. I Have a DREAM! (DiffeRentially privatE smArt Metering). In *Information Hiding*, Tomáš Filler, Tomáš Pevný, Scott Craver, and Andrew Ker (Eds.). Number 6958 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, 118–132.

[3] Ross Anderson and Shailendra Fuloria. 2010. On the Security Economics of Electricity Metering.. In *The Ninth Workshop on the Economics of Information Security (WEIS 2010)*. Citeseer, Harvard University.

[4] Kenneth P. Birman, Robbert van Renesse, and Werner Vogels. 2001. Spinglass: secure and scalable communication tools for mission-critical computing. In *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX '01. Proceedings*, Vol. 2. IEEE, Anaheim, CA, 85–99. https://doi.org/10.1109/DISCEX.2001.932161

[5] Michael Bluejay. 2013. How much electricity does my stuff use? (2013). http://michaelbluejay.com/electricity/howmuch.html

[6] David Chaum. 1985. Security Without Identification: Transaction Systems to Make Big Brother Obsolete. *Commun. ACM* 28, 10 (Oct. 1985), 1030–1044. https://doi.org/10.1145/4372.4373

[7] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '87)*. ACM, New York, NY, USA, 1–12. http://doi.acm.org/10.1145/41840.41841

[8] Tim Dierks and Eric Rescorla. 2008. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. IETF. https://tools.ietf.org/html/rfc5246

[9] Danny Dolev. 1982. The Byzantine Generals Strike Again. *Journal of Algorithms* 3, 1 (March 1982), 14–30. https://doi.org/10.1016/0196-6774(82)90004-9

[10] John R. Douceur. 2002. The Sybil Attack. In *Peer-to-Peer Systems*, Peter Druschel, Frans Kaashoek, and Antony Rowstron (Eds.). LNCS, Vol. 2429. Springer, Berlin, Heidelberg, 251–260.

[11] Cynthia Dwork. 2006. Differential Privacy. In *Automata, Languages and Programming (ICALP)*, Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener (Eds.). LNCS, Vol. 4052. Springer, Berlin, Heidelberg, 1–12. https://doi.org/10.1007/11787006_1

[12] Cynthia Dwork. 2011. A Firm Foundation for Private Data Analysis. *Commun. ACM* 54, 1 (Jan. 2011), 86–95.

[13] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *Advances in Cryptology - EUROCRYPT 2006*, Serge Vaudenay (Ed.). Lecture Notes in Computer Science, Vol. 4004. Springer, Berlin, Heidelberg, 486–503.

[14] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography*, Shai Halevi and Tal Rabin (Eds.). LNCS, Vol. 3876. Springer, Berlin, Heidelberg, 265–284. https://doi.org/10.1007/11681878_14

[15] Christopher Hooley. 1967. On Artin's conjecture. *Journal für die reine und angewandte Mathematik (Crelles Journal)* 225 (1967), 209–220. https://doi.org/10.1515/crll.1967.225.209

[16] Márk Jelasity, Geoffrey Canright, and Kenth Engø-Monsen. 2007. Asynchronous Distributed Power Iteration with Gossip-based Normalization. In *Euro-Par 2007 (LNCS)*, Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol (Eds.), Vol. 4641. Springer, Berlin, Heidelberg, 514–525. https://doi.org/10.1007/978-3-540-74466-5_55

[17] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. 2009. T-Man: Gossip-based fast overlay topology construction. *Computer Networks* 53, 13 (Aug. 2009), 2321–2339. https://doi.org/10.1016/j.comnet.2009.03.013

[18] Gian Paolo Jesi, Alberto Montresor, and Maarten van Steen. 2010. Secure peer sampling. *Computer Networks* 54, 12 (Aug. 2010), 2086–2098. https://doi.org/10.1016/j.comnet.2010.03.020

[19] Håvard Johansen, André Allavena, and Robbert van Renesse. 2006. Fireflies: Scalable Support for Intrusion-tolerant Network Overlays. In *Proc. 1st ACM SIGOPS/EuroSys European Conf. on Comp. Systems 2006 (EuroSys '06)*. ACM, New York, NY, USA, 3–13. https://doi.org/10.1145/1217935.1217937

[20] D. Kempe, A. Dobra, and J. Gehrke. 2003. Gossip-based computation of aggregate information. In *44th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, Cambridge, MA, 482–491. https://doi.org/10.1109/SFCS.2003.1238221

[21] Julia Lane, Victoria Stodden, Stefan Bender, and Helen Nissenbaum (Eds.). 2014. *Privacy, Big Data, and the Public Good*. Cambridge University Press, Cambridge, UK.

[22] Christopher Laughman, Kwangduk Lee, Robert Cox, Steven Shaw, Steven Leeb, Les Norford, and Peter Armstrong. 2003. Power signature analysis. *IEEE Power and Energy Magazine* 1, 2 (March 2003), 56–63. https://doi.org/10.1109/MPAE.2003.1192027

[23] Lawrence Berkeley National Laboratory. 2015. Default Energy Consumption of MELs. (2015). http://hes-documentation.lbl.gov/calculation-methodology/calculation-of-energy-consumption/major-appliances/

miscellaneous-equipment-energy-consumption/default-energy-consumption-of-mels
[24] Fenjun Li, Bo Luo, and Peng Liu. 2010. Secure Information Aggregation for Smart Grids Using Homomorphic Encryption. In *2010 First IEEE International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, Gaithersburg, MD, 327–332. https://doi.org/10.1109/SMARTGRID.2010.5622064
[25] Harry C. Li, Allen Clement, Edmund L. Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. 2006. BAR Gossip. In *Proc. 7th Symp. on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 191–204. http://dl.acm.org/citation.cfm?id=1298455.1298474
[26] Mikhail Lisovich, Dierdre Mulligan, and Steven B. Wicker. 2010. Inferring Personal Information from Demand-Response Systems. *IEEE Security & Privacy* 8, 1 (Jan. 2010), 11–20. https://doi.org/10.1109/MSP.2010.40
[27] Patrick McDaniel and Stephen McLaughlin. 2009. Security and Privacy Challenges in the Smart Grid. *IEEE Security and Privacy* 7, 3 (May 2009), 75–77. https://doi.org/10.1109/MSP.2009.76
[28] Alberto Montresor and Márk Jelasity. 2009. PeerSim: A Scalable P2P Simulator. In *Proc. 9th IEEE Intl. Conf. on Peer-to-Peer Computing (P2P 2009)*. IEEE, Seattle, Washington, USA, 99–100. https://doi.org/10.1109/P2P.2009.5284506
[29] Pieter Moree. 2012. Artin's Primitive Root Conjecture – A Survey. *Integers* 12, 6 (2012), 1305–1416. https://doi.org/10.1515/integers-2012-0043
[30] Róbert Ormándi, István Hegedűs, and Márk Jelasity. 2013. Gossip learning with linear models on fully distributed data. *Concurrency and Computation: Practice and Experience* 25, 4 (Feb. 2013), 556–571. https://doi.org/10.1002/cpe.2858
[31] Jukka V. Paatero and Peter D. Lund. 2006. A model for generating household electricity load profiles. *International Journal of Energy Research* 30, 5 (April 2006), 273–290. https://doi.org/10.1002/er.1136
[32] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology — EUROCRYPT '99*, Jacques Stern (Ed.). Lecture Notes in Computer Science, Vol. 1592. Springer, Berlin, Heidelberg, 223–238.
[33] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. 1998. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications* 16, 4 (May 1998), 482–494. https://doi.org/10.1109/49.668972
[34] Jared Saia and Mahdi Zamani. 2015. Recent Results in Scalable Multi-Party Computation. In *41st International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'15) (Lecture Nodes in Computer Science)*, Giuseppe F. Italiano, Tiziana Margaria-Steffen, Jaroslav Pokorný, Jean-Jacques Quisquater, and Roger Wattenhofer (Eds.), Vol. 8939. Springer, Berlin, Heidelberg, 24–44. https://doi.org/10.1007/978-3-662-46078-8_3
[35] US Census Bureau. 2015. National Summary Tables - AHS 2013. (May 2015). http://www.census.gov/programs-surveys/ahs/data/2013/national-summary-report-and-tables---ahs-2013.html
[36] Robbert van Renesse, Kenneth Birman, Dan Dumitriu, and Werner Vogels. 2002. Scalable Management and Data Mining Using Astrolabe. In *Peer-to-Peer Systems*, Peter Druschel, Frans Kaashoek, and Antony Rowstron (Eds.). Number 2429 in LNCS. Springer, Berlin, Heidelberg, 280–294.
[37] World Economic Forum. 2011. Personal Data: The Emergence of a New Asset Class. (2011). http://www3.weforum.org/docs/WEF_ITTC_PersonalDataNewAsset_Report_2011.pdf
[38] G. Pascal Zachary. 2011. Saving smart meters from a backlash. *IEEE Spectrum* 48, 8 (Aug. 2011), 8–8. https://doi.org/10.1109/MSPEC.2011.5960144

Received November 2016; revised November 2017; accepted March 2018