DStress: Efficient Differentially Private Computations on Distributed Data

Antonis Papadimitriou Arjun Narayan Andreas Haeberlen
University of Pennsylvania

Abstract

In this paper, we present DStress, a system that can efficiently perform computations on graphs that contain confidential data. DStress assumes that the graph is physically distributed across many participants, and that each participant only knows a small subgraph; it protects privacy by enforcing tight, provable limits on how much each participant can learn about the rest of the graph.

We also study one concrete instance of this problem: measuring systemic risk in financial networks. Systemic risk is the likelihood of cascading bankruptcies – as, e.g., during the financial crisis of 2008 – and it can be quantified based on the dependencies between financial institutions; however, the necessary data is highly sensitive and cannot be safely disclosed. We show that DStress can implement two different systemic risk models from the theoretical economics literature. Our experimental evaluation suggests that DStress can run the corresponding computations in about five hours, whereas a naïve approach could take several decades.

1. Introduction

In the age of "big data", it is well known that many interesting things can be learned by collecting and analyzing large graphs, and a number of tools – including GraphLab [47], PowerGraph [35], and GraphX [36] – have been developed to make such analyses fast and convenient. Typically, these tools assume that the user has a *property graph G* (that is, a graph that has some data associated with its vertexes and/or edges) and wishes to compute some function $F(G)$ over this graph and its properties. A common assumption is that there is a single entity that knows the entire graph G and is therefore able to compute $F(G)$ directly.

However, there is another class of use cases where the graph *G contains sensitive information and is spread across multiple administrative domains*. In this situation, each domain knows only a subset of the vertexes and edges, so it cannot compute $F(G)$ on its own, but the domains may not be willing to share their data with each other because of privacy concerns.

One interesting real-world instance of this problem is the computation of *systemic risk* in financial networks [1]. Motivated in part by the financial crisis of 2008, this topic has recently seen a lot of interest in the theoretical economics literature. Briefly, economists have discovered that one of the causes for the crisis was a "snowball effect" in which a few initial bankruptcies caused the failure of more and more other banks due to financial dependencies. *In theory*, it would be possible to quantify the risk of such a cascading failure by looking at the graph of financial dependencies between the banks, and in fact economists have already developed a number of metrics [25, 27] that could be used to quantify this "systemic" risk, and to ideally give some early warning of an impending crisis.

However, *in practice*, the required information is extremely sensitive because it directly reflects the business strategy of each bank. It is so sensitive, in fact, that banks would prefer not to share it even with the government. This is why current audits (such as the annual "stress tests" that were introduced after the crisis, e.g., by the Dodd-Frank Act in the United States) are strictly compartmentalized, so that each auditor is only allowed to look at the data of one particular bank. This provides some basic security, but it is not sufficient to discover complex interdependencies, which would require looking at data from *all* the banks. This is why, in a recent working paper [30], the Office of Financial Research (OFR) has started investigating ways to perform system-wide stress tests while protecting confidentiality.

One possible approach would be to use secure multiparty computation (MPC) [63], which would enable the banks to collectively evaluate a function $F(G)$ over their combined financial data – say, one of the existing systemic risk measures [25, 27]. However, there are two challenges with this approach. The first is performance: MPC does not scale well to large numbers of parties or complex computations. As we will show, computing systemic risk with a straightforward application of MPC would literally take many years.

The second, and somewhat more subtle, challenge is privacy: MPC only guarantees that no one can learn the *intermediate* results of the computation. However, even the *final* result (the value of $F(G)$) can reveal information about the

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064218
Reprinted from EuroSys '17,, [Unknown Proceedings], April 23 - 26, 2017, Belgrade, Serbia, pp. 1–15.

underlying graph G, especially in the presence of auxiliary information. To see this intuitively, imagine using MPC to compute the average weight of the people in a room. This will not reveal the weight of any single individual, but the adversary can still infer the presence of a team of sumo wrestlers by looking only at the final result. A similar concern arises in the context of financial data [30, §4.2].

In this paper, we present a system called DStress that can efficiently analyze graphs that are spread across thousands of administrative domains, while giving strong, provable privacy guarantees on *both* the topology of the graph G *and* the data it contains. DStress supports *vertex programs*, a programming model that is also used in Pregel [48] and Graphlab [47], two popular frameworks for non-confidential graph computations. It addresses the first challenge with a special graph-computation runtime that can execute vertex programs in a distributed fashion, using MPC and a variant of ElGamal encryption for transferring data between domains, and it addresses the second challenge by keeping intermediate results encrypted at all times, and by offering differential privacy [22, 24] on the final result.

We have built a prototype of DStress, and we have evaluated it using two systemic-risk models from the theoretical economics literature. Our results show that these models could be evaluated on the entire U.S. banking system in less than five hours on commodity hardware, using about 750 MB of traffic per bank. In our technical report [57], we also show that the use of differential privacy (which was already suggested by the OFR working paper [30]) does not significantly diminish the utility of the systemic risk measure. In summary, this paper makes the following three contributions:

- DStress, a scalable system for graph analytics with strong privacy guarantees (Section 3);
- an application of DStress to privately measuring systemic risk in financial networks (Section 4); and
- an experimental evaluation, based on a prototype implementation of DStress (Section 5).

2. Overview

We consider a scenario with a group of N participants P_i, $i = 1,\ldots,N$ that each know one vertex v_i of a directed graph G, as well as a) the edges that begin or end at v_i, and b) any properties associated with v_i. The participants wish to collectively compute a function $F(G)$, such that:

- **Value privacy:** The computation process does not reveal properties of v_i to participants other than P_i;
- **Edge privacy:** The computation process does not reveal the presence or absence of an edge (v_i, v_j) to participants other than i and j; and
- **Output privacy:** The final output $F(G)$ does not reveal too much information about individual vertexes or edges in G.

In the scenarios we are interested in, the number of parties N is on the order of thousands; for instance, the number of major banks in the U.S. banking system is about $N = 1,750$.

2.1 Background: Systemic risk

To explain how financial networks fit this model, we now give a very brief introduction to systemic risk. Since this is a complex topic, we focus on the aspects that are most relevant to DStress; for more details, see [9, 13, 30].

Banks and other financial institutions, as part of their regular business with clients, are exposed to *risk*. We can think of this risk as a specific abstract event x_i – such as a drop in house prices – that, if it happened, would cause bank b to lose a certain amount of money y_i. Thus, b's *balance sheet* has *exposure* of the form: $(\texttt{if } x_i \texttt{ then } -y_i)$. To prevent a buildup of excess exposure to any single future event, b can in advance create *derivatives* on event x_i and sell part of this exposure to other banks (presumably for a fee). We can think of these derivatives as "insurance contracts" that specify that a certain sum will be due if and when a particular event occurs. Thus, if b bought "insurance" against x_i from another bank that pays z_i, b's exposure would now be: $(\texttt{if } x_i \texttt{ then } z_i - y_i)$. More complicated forms of derivatives also exist.

Banks regularly reinsure their risk by buying additional derivatives from other banks. The result is a network of dependencies that spans the entire financial sector. We can think of this as a graph G that contains a vertex for each bank and an edge (b_1, b_2) whenever b_1 has sold a derivative to b_2. Vertexes would be annotated with the liquid cash reserves of the corresponding bank, and edges would be annotated with the payment that is due in each event. Using this graph, it is possible to essentially simulate what would happen if a particular event were to occur – including possible cascading failures, where the initial bankruptcy of a few critical banks causes a "domino effect" that eventually affects a large fraction of the network. The expected "damage" (and thus the systemic risk) can then be measured in a variety of ways – e.g., as the amount of money the government would need to inject in order to stabilize the system. In Section 4, we discuss two concrete models from the economics literature in more detail.

2.2 Strawman solutions

One obvious way to compute the systemic risk would be to create an all-powerful government regulator that has access to the financial data of all the banks. However, this does not seem practical, since banks critical rely on secrecy to protect their business practices [1]. Currently, regulatory bodies that deal with *less* sensitive information about *individual* banks already have extremely restrictive legal safeguards and multiple levels of oversight [30, §3.1].

Another potential approach, first suggested by [1], would be to use multi-party computation (MPC) [63]: one could design a circuit that takes each bank's books as inputs, exe-

cutes the simulation in MPC, and finally outputs the desired measure of risk. This approach would be more palatable for the banks, since they would not need to reveal their secret inputs. However, the circuit would be enormous: even the simplest models of contagion in the literature essentially require raising a $N \times N$ matrix to a large power, where N is the number of banks (i.e., about 1,750). Despite recent advances in MPC, such as [12, 15, 17, 65], evaluating such a large circuit with $N = 1,750$ parties is far beyond current technology.

The cost of MPC could be reduced somewhat by delegating the computation to a smaller number of parties, as in Sharemind [12] or PICCO [65]. However, this approach would do nothing to reduce the size of the circuit, and, given the high stakes involved, the number of parties would still need to be large – the largest collusion case reported in the literature involved 16 banks [60]!

As discussed earlier, none of these approaches would provide output privacy, and this would be a serious concern, since the final output (i.e., the current level of systemic risk) could be enough for some of the banks to make inferences about the graph, particularly if they already know some of the other edges and vertexes.

2.3 Our approach

Our approach is based on two key insights. Our first observation is that much of the enormous cost of the MPC-based strawman comes from the fact that the graph is itself confidential and therefore must be an input to the computation. We can get around this by formulating the function F as a *vertex program* – that is, as a sequence of computations at each vertex that are interleaved with message exchanges over the edges – and by executing it in a distributed fashion. This is safe because each participant already knows the edges that are adjacent to her vertex; the main challenge is to prevent information leakage through intermediate results. In DStress, we accomplish this with a combination of secret sharing, small MPC invocations for the computations at each vertex, and a special protocol for transferring shares without revealing the topology of the graph.

Our second key insight is that we can use *differential privacy* [24] to achieve output privacy. Differential privacy provides strong, provable privacy guarantees, which should be reassuring to the banks. Its main cost is the addition of a small amount of random noise to the output, but, since we are looking for early warnings of large problems, a bit of imprecision (e.g., a shortfall of $1 billion is reported as $0.95 billion) should not affect the utility of the results. If a potential problem is detected, a more detailed investigation could be conducted outside of our system.

3. The DStress system

We begin by briefly reviewing three technologies that DStress relies on: differential privacy, secure multiparty computation, and ElGamal encryption.

Differential privacy: DStress is designed to provide *differential privacy* [24], one of the strongest known privacy guarantees. Differential privacy has a number of features that are attractive in our setting, such as protection against attacks based on auxiliary data (which have been the source of several recent privacy breaches [3, 6, 53]), strong composition theorems, and a solid mathematical foundation with provable guarantees.

Differential privacy is a property of *randomized* queries – i.e., the query computes not a single value but rather a probability distribution over the range R of possible outputs, and the actual output is then drawn from that distribution. This can be thought of as adding a small amount of noise to the output. Intuitively, a query is differentially private if a small change to the input only has a statistically negligible effect on the output distribution.

More formally, let I be the set of possible input data sets. We say that two input data sets $d_1, d_2 \in I$ are similar (and we write $d_1 \sim d_2$) if they differ in at most one element. Then, a randomized query q with range R is ε-differentially private if, for all possible sets of outputs $S \subseteq R$ and all input data sets $d_1, d_2 \in I$ with $d_1 \sim d_2$,

$$Pr[q(d_1) \in S] \leq e^{\varepsilon} \cdot Pr[q(d_2) \in S].$$

That is, any change to an individual element of the input data can cause at most a small multiplicative difference (e^{ε}) in the probability of *any* set of outcomes S. The parameter ε controls the strength of the privacy guarantee; smaller values result in better privacy. For more information on how to choose ε, see, e.g., [40].

A common way to achieve differential privacy for queries with numeric outputs is the *Laplace mechanism* [24], which works as follows. Suppose $\bar{q} : I \to R$ is a deterministic, real-valued function over the input data, and suppose $\bar{q}$ has a finite *sensitivity* s to changes in its input, i.e., $|\bar{q}(d_1) - \bar{q}(d_2)| \leq s$ for all similar databases $d_1, d_2 \in I$. Then $q := \bar{q} + Lap(s/\varepsilon)$, i.e., the combination of $\bar{q}$ and a noise term drawn from a Laplace distribution with parameter s/ε, is ε-differentially private. This corresponds to the intuition that the more sensitive the query, and the stronger the desired guarantee, the more "noise" is needed to achieve that guarantee.

Secure multiparty computation: DStress relies on secure multiparty computation (MPC) to perform certain steps of the graph algorithm it is running. MPC is a way for a set of mutually distrustful parties to evaluate a function f over some confidential input data x, such that no party can learn anything about x other than what the output $y := f(x)$ already implies. In the specific protocol we use (GMW [34]), each party i initially holds a *share* x_i of the input x such that $x = \oplus_i x_i$ (in other words, the input can be obtained by XORing all the shares together), and, after the protocol terminates, each party similarly holds a share y_i of the output $y = f(x)$. The function f itself is represented as a Boolean circuit. GMW is *collusion-resistant* in the sense that, if $k+1$

parties participate in the protocol, the confidentiality of x is protected as long as no more than k of the parties collude.

ElGamal encryption: For reason that will become clear later, DStress requires an encryption scheme with two unusual properties: an additive homomorphism and a way to re-randomize public keys. Both can be elegantly accomplished using a variant of ElGamal [26]. The original ElGamal scheme consists of three functions: a key generator, an encryption and a decryption function. These functions are defined over some cyclic group G. Assume G is of order q and has a generator $g \in G$. ElGamal's key generator function returns a random element $x \in Z_q$ as the secret key, and a public key $h = g^x$. Moreover, given a public key h and a message m, the encryption function picks a random $y \in Z_q$ (sometimes called an ephemeral key) and returns the ciphertext $c = (g^y, m \cdot h^y) = (g^y, m \cdot g^{xy})$. Given a ciphertext $c = (c_1, c_2)$, and a secret key x, the decryption function first computes $s = c_1^x = g^{xy}$, then $s^{-1} = c_1^{(q-x)} = g^{y(q-x)}$ and finally returns the recovered plaintext as $c_2 \cdot s^{-1} = m g^{xy} g^{qy-xy} = m g^{qy} = m$.

ElGamal itself has a multiplicative homomorphism: if we encrypt two messages m_1 and m_2 and multiply the two ciphertexts together, the result decrypts to the product $m_1 \cdot m_2$. However, this can be turned into an additive homomorphism using a small trick, which is to encrypt not the message m itself but rather g^m. The resulting scheme (exponential ElGamal [19]) ensures that the product of two ciphertexts $g^{m_1} \cdot g^{m_2} = g^{m_1+m_2}$ now decrypts to the sum of the underlying messages. The downside is that there is no easy way to go back from g^m to m – but, if the number of valid messages is small enough, the recipient can use a lookup table to decrypt: simply precompute g^c for all candidate messages c and compare the results to the g^m she received. Exponential ElGamal also satisfies our second requirement: if g^x is a public key, we can re-randomize it by raising it to some value r, yielding a new public key g^{xr}. If a message is encrypted with this new public key, it will not decrypt with the original private key x; however, this can be fixed by raising the ephemeral key in the ciphertext to r as well. Notice that both operations (re-randomizing the public key and adjusting ciphertexts) can be performed without knowledge of the private key x.

3.1 Programming model

DStress is designed to run *vertex programs*. A vertex program consists of (1) a graph $G := (V, E)$; (2) for each vertex $v \in V$, an initial state s_v^0 and an update function f_v; (3) a number of iterations n; (4) an aggregation function A; (5) a no-op message $\perp$; and (6) a sensitivity s. DStress executes such an algorithm as follows. First, each vertex v is first set to its initial state s_v^0. Next, DStress performs a *computation step* by invoking the update function f_v for each vertex, which outputs a new state s_v^1 and, for each neighbor of v in G, exactly one message. (If more messages need to be sent, they can be included in one larger message.) When v has no data to send to some neighbor, it outputs the no-op message $\perp$ instead; this is necessary to avoid leaking information through its communication pattern. The computation step is followed by a *communication step*, in which each vertex sends its messages along the edges to its neighbors; the recipients then use the messages as additional inputs for their next computation step. After n computation and communication steps, DStress performs a final computation step and then invokes the aggregation function A, which reads the final state of each node and combines the states into a single output value. Finally, DStress draws a noise term from a Laplace distribution $Lap(s/\varepsilon)$ and adds it to the output value, which yields result of the computation.

The vertex programming model is quite general; there are other systems that implement it – such as Pregel [48] – and it can express a wide variety of graph algorithms. Not all of these algorithms have privacy constraints, but there are many that do; for instance, cloud reliability [64], criminal intelligence [43, 62], and social science [14, 28, 45] all involve analyzing graphs that can span multiple administrative domains.

3.2 Threat model and assumptions

DStress relies on the following five assumptions:

1. The nodes are honest but curious (HbC), i.e., they will faithfully execute DStress but try to learn as much about the graph as they can;
2. The nodes do not have enough computational power to break the cryptographic primitives we use;
3. There is an upper bound k on the number of nodes that will collude;
4. There is a (publicly known) upper bound D on the degree of any node in the graph; and
5. There is a trusted party (TP) that knows the identities of all the nodes in the system and can perform some simple setup steps. (The TP can be offline and never sees any private information.)

Assumption 1 may seem counterintuitive at first, especially in our systemic-risk case study, which involves valuable financial data. However, recall that banks are already heavily audited and inspected in most countries. These audits are compartmentalized, so they cannot be used to measure systemic risk directly; however, they could certainly be used to verify that each bank has input the correct data and has executed DStress correctly.

Assumption 2 is standard for virtually all protocols that use cryptography; it implies that DStress offers *computational* differential privacy [50]. Assumption 3 seems plausible for the banking scenario because of antitrust laws that prevent large-scale collusion between banks; for other applications, the bound k could be chosen based on the largest observed instance of collusion, plus a safety margin. Assumption 4 is in accordance to economic incentives described

in [18], which suggest that the financial network is not fully connected. An example of an institution that satisfies assumption 5 is the Federal Reserve.

3.3 Basic operation

When executing an algorithm, DStress runs on a distributed set of *nodes*, and it maps each vertex in the graph to a specific node that will provide the initial state for that vertex and that will coordinate the corresponding computation and communication steps. Unlike the (logical) vertexes, which communicate over edges in the graph, the (physical) nodes can communicate directly over a network, such as the Internet. We expect that, for privacy reasons, each participant would want to operate its own node, so that it does not have to reveal its initial state to another party.

To prevent privacy leaks, DStress must not allow any node to see intermediate states of the computation – not even that of their own vertex, since it may have changed based on messages from other vertices. Hence, DStress associates each node i with a set B_i of other nodes that we refer to as the *block* of i. The members of the block each hold a share of the vertex's current state, and they use MPC to update the state based on incoming messages. To prevent colluding nodes from learning the state of a vertex, each block contains $k+1$ nodes, where k is the collusion bound we have assumed.

A key challenge is to enable vertices to communicate without weakening security or revealing the structure of the graph. If (i, j) is an edge in the graph and i wants to send a message m to j, then the members of block B_i, who would each hold a share of m after the MPC that generated it, cannot simply send their shares of m to the members of block B_j, since that would reveal the existence of the edge (i, j) to both blocks. To avoid this problem, DStress redirects all communication between blocks B_i and B_j through the nodes i and j, who already know that an edge exists between them. To prevent i and j from reconstructing m, all shares are encrypted, and we use ElGamal's key re-randomization feature to prevent the senders from learning the identities of the recipients through their public keys.

3.4 One-time setup step

Before DStress can be used with a new graph, it is necessary to perform a one-time setup step. This step has two different purposes. First, it associates each node i with a block B_i that contains $k+1$ *different* nodes, including i. This is necessary to prevent curious nodes from filling their own blocks with Sybil identities or with multiple instances of the same node, which would weaken DStress's collusion-resistance. Second, it equips each block B_i with D different sets of public keys. This is necessary to prevent colluding neighbors of i from identifying members of B_i based on their public keys.

The setup step is coordinated by the trusted party (TP). The TP begins by asking each node i for a) i's public ElGamal key, and b) D different *neighbor keys* $n^i_1, \ldots, n^i_D$, which i can choose arbitrarily from Z_q. The TP then randomly picks a list of members for i's block B_i and publishes $\sigma_{\text{TP}}(\{(k, B_k)\}_{k=1..|V|}, B_A)$ – that is, a list of nodes and their blocks that is signed with the TP's private key. (Note that this list does not contain information about edges and thus reveals nothing about the structure of the graph.) B_A is a special block that is used for aggregation (Section 3.6); its $k+1$ members are also chosen randomly by the TP.

Next, the TP generates D *block certificates* for each block B_i. A block certificate is a tuple $C_{i,j} := \sigma_{\text{TP}}(g^{x_1 n^i_j}, g^{x_2 n^i_j}, \ldots, g^{x_D n^{i,j}})$, that is, it contains one public key for each member of the corresponding block, but the keys in the j.th certificate for node i are re-randomized using the j.th neighbor key from node i. The TP signs each of the D block certificates and then sends them to node i, who forwards each certificate to a different neighbor. (If i has fewer than D neighbors in the graph, it simply discards the leftover certificates.) Finally, each node i distributes the block certificates it has received from its neighbors to the members of its own block B_i, but without identifying the specific neighbor – i only tells the members of B_i which certificate is for its first neighbor, its second neighbor, and so on.

Once this step is completed, the TP is no longer needed and can leave the system. Notice that the TP has never learned the topology of the graph.

3.5 Message transfer protocol

Next, we describe how the block certificates can be used to securely send messages along edges of the graph. Since the details of the full protocol are somewhat complicated, we start with a simple but flawed strawman protocol and then derive the full protocol in several steps.

Recall that, at the end of each computation step, each node i sends a message $m_{i,j}$ (possibly the no-op message $\perp$) to each neighbor j in the graph. Since the computation step is performed in MPC, at the end of the step each member of i's block B_i holds one share of $m_{i,j}$, such that the message can be reconstructed by XORing all the shares together. These shares must be transferred to the members of B_j, who then use them as inputs to j's' next computation step.

Strawman #1: *Each $x \in B_i$ picks a different public key from j's block certificate and encrypts its share s_x of $m_{i,j}$ with this key. Then the members of B_i forward their encrypted shares to i, who forwards them to j. j adjusts the ephemeral keys in the ciphertexts using the neighbor key $n_{i,j}$ and then forwards them to the members of B_j, who decrypt them and thus each obtain one share of $m_{i,j}$.*

This approach prevents the members of B_i and B_j from learning about each other directly: all communication is via i and j, and the members of B_i only see the re-randomized public keys of the members of B_j, so they cannot identify the latter by recognizing their public keys. However, this approach weakens collusion resistance: if the same node n happens to be a member of both B_i and B_j, or if two nodes

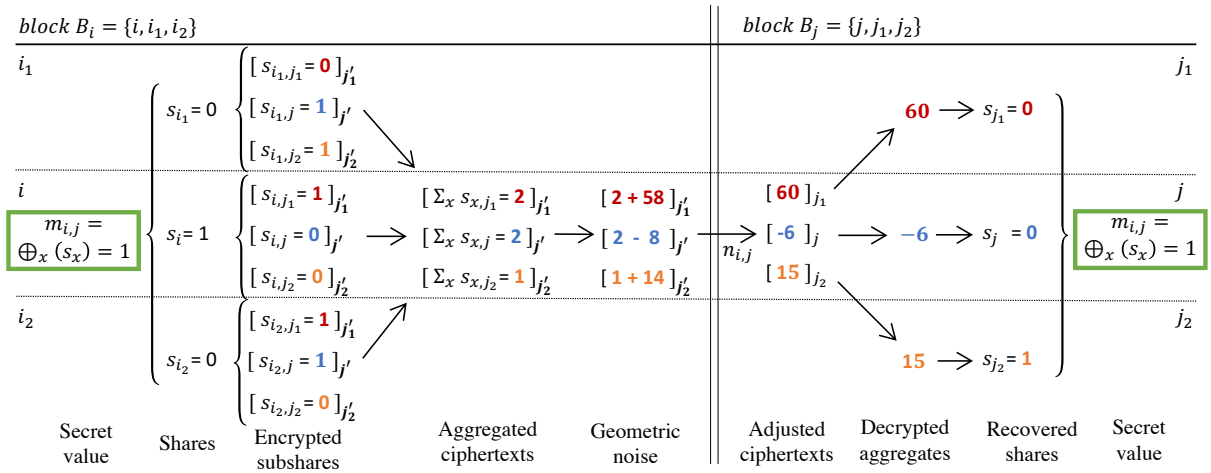

Figure 1: A message transfer example between two blocks of three nodes. $[\ldots]_{j'_x}$ denotes a ciphertext encrypted with the randomized public key of j_x, and $[\ldots]_{j_x}$ denotes a ciphertext encrypted with the original key.

$n_1 \in B_i$ and $n_2 \in B_j$ collude, they can potentially learn *two* shares. To prevent this, we make the following change:

Strawman #2: *Like strawman #1, except that each $x \in B_i$ splits its share s_x into $k+1$* subshares *$s_{x,1}, \ldots, s_{x,k+1}$ such that $s_x = \oplus_{y=1..k+1} s_{x,y}$ and then encrypts a different subshare for each member of B_j.*

As long as the secret-sharing scheme is associative and commutative, the members of B_j can obtain valid (but different) shares of $m_{i,j}$ simply by combining all the subshares they receive. This change also restores collusion resistance: as long as both B_i and B_j have at least one member that does not collude, the colluding nodes will always miss at least one share, namely the one that is sent between the two non-colluding nodes. However, if $n_1 \in B_i$ and $n_2 \in B_j$ collude, they can still infer the presence of edges by recognizing subshares: n_1 can use some external channel to tell n_2 about the subshares it has sent, and if n_2 subsequently receives one of them, they can infer that their blocks are connected by an edge. We fix this as follows:

Strawman #3: *Like strawman #2, except that each member of B_i breaks its subshare into individual bits and encrypts each bit separately. The encrypted bits are forwarded through i as before, but, rather than forwarding them to j directly, i uses the homomorphic addition in exponential ElGamal to combine the corresponding bits from different subshares. This yields the encrypted sum of bits of the shares, which j then forwards to B_j. Members of B_j decrypt the sums and set their bit share to 0 iff the sum is even.*

This approach *almost* meets our requirements, since the recipients never see the senders' original subshares and thus cannot recognize them. However, because the homomorphic operation is an addition and not an XOR, the encrypted "bits" that arrive at B_j are actually numbers that correspond to the number of ones in the original subshares. This leaks some information about the original shares. To see why, consider the (extreme) case where the adversary controls k of the $k+1$ nodes in both B_i and B_j, and wishes to learn whether the edge (i, j) exists. Suppose that, during some particular communication step, the subshares of the adversary's nodes in B_i for each of the D messages add up to $S_1, \ldots, S_D$, respectively. If the adversary's nodes in B_j then receive only sets of shares that add up to less than $S_n - 1$ or more than $S_n + 1$ for all $1 \leq n \leq D$, then the adversary knows that none of the received messages could have come from B_i, so the edge (i, j) cannot exist. Conversely, if, over the course of many communication steps, the adversary's nodes in B_j always receive some set of shares that add up to $S_n \pm 1$ for some n, the adversary can be increasingly confident (though never completely certain) that the edge (i, j) does exist. If the adversary controls fewer than k of the $k+1$ nodes in B_i and/or B_j, the risk of the adversary learning something about (i, j) in any particular communication step diminishes, but it never completely disappears. We mitigate this risk by making one final change:

Final protocol: *Like strawman #3, except that i homomorphically adds an even random number from $2 \cdot Geo(\alpha^{\frac{2}{k+1}})$ to each encrypted bit before forwarding it to B_j via j, where Geo is the geometric distribution*[1] *as described in [33] and α is a parameter in $(0, 1)$.*

This preserves correctness: the recipients will receive an even (but otherwise random) number if and only if it would have received a zero bit using strawman #3. However, the adversary's chances of learning something useful have diminished dramatically: the sum of bits is now noised, and it is very hard for the adversary to extract information from this side-channel. In fact, as [33] shows, the application of geometric noise provides ε-differential privacy, where $\varepsilon = -\ln \alpha$. This way, we can maintain a privacy budget

[1] The geometric distribution is a discretized version of the Laplace distribution, which is widely used in differential privacy.

to keep track of what the adversary learns and make sure that the probability of an edge leaking is minimal. For details regarding the sensitivity analysis and differential privacy guarantees of the protocol, please refer to our technical report [57].

The overall effect is that, for each edge (i, j) in the graph, the members of block B_i can transfer the shares of a message to the members of block B_j such that a) no group of k or fewer colluding nodes can learn the contents of the message, and that b) edge privacy (Section 2) is maintained. We provide a formal proof of the first property and a detailed discussion of edge privacy in our technical report [57].

3.6 Executing a program

Next, we describe how DStress executes a program. For simplicity, we focus only on the algorithm and ignore practical challenges, such as fault tolerance; these challenges are orthogonal and can be addressed with existing techniques. Recall that each execution has n computation and communication steps, followed by aggregation and noising.

Initialization step. DStress maintains the invariant that, at the beginning of each computation step, each member of a node i's block B_i has 1) a share of the current state of i's vertex, and 2) shares of D input messages, which can either be messages of i's neighbors or instances of the no-op message $\perp$. To make this invariant true at the first step, each node i starts by loading the initial state of its local vertex, as well as D copies of $\perp$ (since there are no real messages yet), and splits each of them into $|B_i|$ shares, one for each member of its block B_i.

Computation step. In each computation step, the members of each block B_i use MPC to evaluate the update function of the corresponding vertex v_i. The circuit has inputs for the D input messages and the current state of v_i, as well as outputs for D output messages and the new state of v_i. (If the degree of the vertex is less than D, some of the messages are copies of $\perp$.) Note that both inputs and outputs of an MPC step remain shared among the members of the block and are never revealed to any individual node.

Communication step. In this step, DStress invokes the protocol from Section 3.5 to send each message along the corresponding edge of the graph. Because each directed edge is used to send exactly one message, a node can immediately proceed to the next computation step once it has received a message from each of its in-neighbors; there is no need for global coordination.

Aggregation+noising step. Once n computation and communication steps have been performed, each block B_i holds shares of vertex v_i's final state. Next, DStress evaluates the aggregation function A on the final states, using the special aggregation block B_A. Each block sends its state shares and some random shares to B_A; the members of B_A then use MPC to a) evaluate A on the states; b) combine the random shares to get a random input seed, c) draw a noise term from $\text{Lap}(s/\varepsilon)$ using the seed; and d) output the sum of that term and the result of A. The simplest way to implement this is to use a single aggregation block, but this could become a bottleneck for larger graphs; in this case, the aggregation can be performed hierarchically, using a tree of aggregation blocks.

3.7 Limitations

DStress currently executes a fixed number of iterations. Dynamic convergence checks are problematic from the perspective of differential privacy because the number of rounds is itself disclosive and would need to be treated as an additional output. However, if the number of rounds is chosen conservatively, this restriction will cost some performance but should not affect correctness.

DStress is limited to executing vertex programs that a) can be expressed as Boolean circuits, and b) have a known, finite sensitivity bound. The first limitation exists because DStress uses MPC to execute the computation steps; it effectively means that the update functions cannot have dynamic loop bounds or unbounded recursion. The second limitation currently prevents ad-hoc queries, but we speculate that DStress could be augmented with automated sensitivity inference, e.g., using linear type systems [31, 38] or a system like CertiPriv [4]. Sensitivity inference for graphs is considered challenging, but the community is making progress with systems like wPINQ [58], which can automatically derive the sensitivity for an important class of graph algorithms. Also, one can find algorithms with known sensitivity in the differential privacy literature (e.g., in [42]), as we did for the two algorithms we used in section 4.4.

DStress's current design assumes a single bound D on the degree of each vertex in the financial network. If, despite the evidence in [18], the maximum degree was very large, this would slow down our algorithm. However, one could avoid this by dividing the vertexes into buckets based on their approximate degree – e.g., one bucket for vertexes with fewer than 100 neighbors and another for the rest. This would reveal a small amount of information about the degree of each bank (which would probably be heavily correlated with the bank's size), but in return, the MPC block computations for most banks would be much faster than if a single conservative degree bound were used for all banks.

4. Case studies

In this section, we describe two different models of financial contagion from the economics literature, and we show how they can be implemented in DStress to compute a measure of systemic risk.

4.1 Metrics and Privacy Guarantees

We follow a recommendation from the OFR working paper [30, §4.3] and measure systemic risk as the *total dollar shortfall (TDS)* – that is, the amount of extra money that the government would need to make available to prevent failures if the contracted event were to occur. It would perhaps

```
--INIT(i)
cash[i]         = Liquid reserve at i
debts[i][j]     = Debt owed by i to j
credits[i][j]   = Debt owed by j to i
totalDebt[i]    = sum_j(debts[i][j])
prorate[i]      = 1.0
noOpMessage     = 0
sensitivity     = 1/r # See Section 4.4


--UPDATE(i)
liquid = cash[i]
foreach j in neighbors(i)
  shortfallJ = recvFrom(j)
  liquid += credits[i][j] - shortfallJ
if (liquid < totalDebt[i])
  prorate[i] = liquid / totalDebt[i]


--COMMUNICATE_WITH(i)
foreach j in neighbors(i)
  sendTo(j, debts[i][j]*(1-prorate[i]))

--AGGREGATE
totalShortfall =
  sum_i(totalDebt[i]*(1-prorate[i]))
```

(a) Based on Eisenberg and Noe [25]

```
--INIT(i)
base[i]         = Base assets held by i
origVal[i]      = initial valuation of i
value[i]        = current valuation of i
insh[i][j]      = share of j held by i
threshold[i]    = i's failure threshold
penalty[i]      = penalty if <threshold
noOpMessage     = 0
sensitivity     = 2/r # See Section 4.4

--UPDATE(i)
value[i] = base[i]
foreach j in neighbors(i)
  discount = recvFrom(j)
  value[i] += insh[i][j] *
      (1-discount)*origVal[i][j]
if (value < threshold[i])
  value[i] = value[i] - penalty[i]

--COMMUNICATE_WITH(i)
foreach j in neighbors(i)
  sendTo(j, 1-(value[i]/origVal[i]))

--AGGREGATE
totalShortfall =
  sum_i((value[i]<threshold[i]) ?
  (threshold[i]-value[i]) : 0)
```

(b) Based on Elliott, Golub, and Jackson [27]

Figure 2: Two algorithms for measuring systemic risk from the economics literature, implemented in DStress.

be more intuitive to compute the number of banks that fail, but TDS has two key advantages. First, it is more meaningful because it can distinguish between a small shortfall such as $\$10,000$ (which, in a large bank, is easily fixed) and a large, more serious shortfall such as \$10 billion. Second, it is a better fit for differential privacy. It is well known that the answer to many questions about graph-shaped data can change radically when even a single edge is added or removed, and thus such questions cannot be answered with differential privacy. However, the TDS is an exception: adding or removing edges does not disproportionally affect the TDS [39].

The privacy guarantee that results when we add noise to the TDS is called *dollar*-differential privacy; it was first introduced in [30]. In this model, the sensitive data we are protecting consists of the investment portfolios of all the banks, and we consider two data sets d_1, d_2 to be similar ($d_1 \sim d_2$) if one can be transformed into the other by reallocating at most T dollars in a single portfolio. The resulting privacy guarantee completely protects all positions with value up to T, although an adversary might be able to infer some information about very large positions.

4.2 The Eisenberg-Noe Model

Our first model, from Eisenberg and Noe [25], considers banks holding debt contracts from and to other banks. A stress test based on this model would first, based on some hypothetical future scenario[2], compute a netted exposure graph on a bilateral basis between the banks, as is done in per-bank stress tests today. After computing each bank's contractual obligations, this would result in a graph of payments between the banks. Then, each bank's liquid reserves plus incoming payments (i.e., debts paid by other banks) would be compared to its total debt. If the debts are bigger, the bank would be deemed bankrupt, and its payments would be adjusted based on what assets the bank actually has. As proven in [25], if there are n banks, this process converges to a unique solution after at most n iterations.

Figure 2(a) shows an implementation in DStress. Initially, the algorithm assumes that each node can pay its obligations in full (`prorate=1`); in each update step, each node i computes its local shortfall as a fraction of its debt, and sends a message to each adjacent node j that contains the amount of i's debt to j that i is unable to pay. The final aggregation step computes the TDS.

4.3 The Elliott-Golub-Jackson Model

The second model, from Elliott, Golub, and Jackson [27], describes a very different type of contagion, using equity cross-holdings to represent inter-institutional dependencies. In this model, there is a set of *primitive assets*, which have

[2] Regulators choose one or more hypothetical events/shocks, and they build custom models based on those described in the economics literature [29].

associated prices. Banks own their own individual basket of these assets, as well as potentially equity in each other. Thus, the valuation of a bank is a) the value of its own primitive assets, plus b) its fraction of the primitive assets owned by other banks in which it (directly or transitively) holds an equity stake. The latter can be computed via fixpoint iteration. The model has another unusual feature: when a bank's valuation falls below a bank-specific threshold, it is considered to have failed, and its value drops by an additional penalty. This is different from the Eisenberg-Noe model, which is inspired by the allocation of assets in traditional bankruptcy proceedings; the intent is to represent "distressed" institutions that may not fail to the point of actual bankruptcy but still face sudden additional costs due to, e.g., a downgraded credit rating.

[27] also shows that the fixpoint is not unique and depends on the starting conditions and on which nodes fail first; thus, there is a possibility of false negatives. However, this is not due to our implementation in DStress – it is simply how the algorithm works as originally proposed. The algorithm is also *not* guaranteed to converge after n steps, as each step can cause a valuation drop even beyond the discontinuous drop. However, as shown in [39], it converges to its final value monotonically, and thus a limited number of iterations provides a good approximation result.

Figure 2(b) shows an implementation in DStress. Initially, each bank has some exogenous valuation `origVal`; in each step, each bank computes a discount to its own value, based on its primitive assets and the current valuation of its equity holdings, and then propagates that discount to its neighbors in the graph. The final aggregation step computes the TDS of all failed banks relative to their failure threshold, as suggested by [27].

4.4 Sensitivity bounds

Recall that DStress requires the programmer to provide a bound on the program's sensitivity to changes in its input. We rely on a proof by Hemenway and Khanna [39], which shows that the sensitivity of the Elliott-Golub-Jackson algorithm is $2/r$, where r is an upper bound on the leverage ratio of the banks (that is, the ratio between a bank's total assets and its equity may not exceed $1:r$). This type of constraint is already mandated by law today because leverage limits provide some stability: they create a "cushion" that can absorb some losses. The proof does not directly consider Eisenberg-Noe, but, using an argument analogous to [39, §5.2], it is possible to derive a sensitivity bound of $1/r$.

4.5 Utility

This leaves two practical questions: 1) how frequently could these algorithms be safely executed, and 2) how does the addition of noise affect the utility of the output? We cannot hope to give final answers here because of the many policy decisions that would be involved, but we can at least provide ballpark figures.

First, we need to choose the privacy parameter ε. We assume that the banks would want to prevent an adversary from increasing their confidence in any fact about the input data by more than a factor of two; this yields $e^{\varepsilon_{\max}} = 2$ and thus a privacy "budget" of $\varepsilon_{\max} = \ln 2$.

Next, we need to calculate the amount of noise that would be added to the output. This depends on a) which input data sets would be considered similar (i.e., at what threshold T the banks would wish to protect their financial data), and b) the sensitivity of the program. For a), we follow an argument from Flood et al. [30] and assume that a granularity of $T = \$1$ billion – roughly the size of the 100th largest bank's equity – is reasonable. For b), we use Elliot-Golub-Jackson as an example and set the leverage bound to $r = 0.1$, as mandated by the Basel III framework [2]. This yields a sensitivity of $2/r = 20$ (independent of the number of iterations); thus, the noise would be drawn from $T \cdot Lap(20/\varepsilon_{\text{query}})$.

Finally, we need to decide how precise the output needs to be, which controls the "privacy cost" $\varepsilon_{\text{query}}$ of the program. In 2015, the annual stress test mandated by Dodd-Frank yielded a TDS of about $500 billion [11], which was considered safe. We add a generous safety margin and assume that it would be sufficient to compute the TDS to within $\pm\$200$ billion. To ensure that the noise is lower than that with at least 95% confidence, we must choose $\varepsilon_{\text{query}} \geq 0.23$.

Since banks must retrospectively disclose their aggregate positions every year anyway, it seems reasonable to replenish the privacy budget once per year. Thus, it seems safe to execute Elliot-Golub-Jackson up to $(\ln 2)/0.23 \approx 3$ times per year, which is more frequent than today's annual stress tests.

4.6 Threat model

Recall from Section 3.2 that DStress assumes that the parties are honest but curious (HbC). At first glance, it is not obvious that this assumption holds universally in the financial world. However, recall that banks are heavily regulated, and that they already have to submit to audits today. It should be possible to use these audits to verify that the banks a) contribute accurate information, and that they b) correctly perform the steps of the DStress algorithm. For privacy reasons, today's audits are compartmentalized – that is, each auditor gets to see only the information from a single bank – but this is sufficient for our purposes: each auditor only needs to verify the steps that are taken by the specific bank she is responsible for,

5. Evaluation

In this section, we report results from our experimental evaluation of DStress. Our main goal is to determine whether DStress's costs are low enough for our application scenario, and whether it is sufficiently scalable.

5.1 Prototype and experimental setup

For our experiments, we built a prototype of DStress that consists of three components: 1) the Wysteria MPC run-

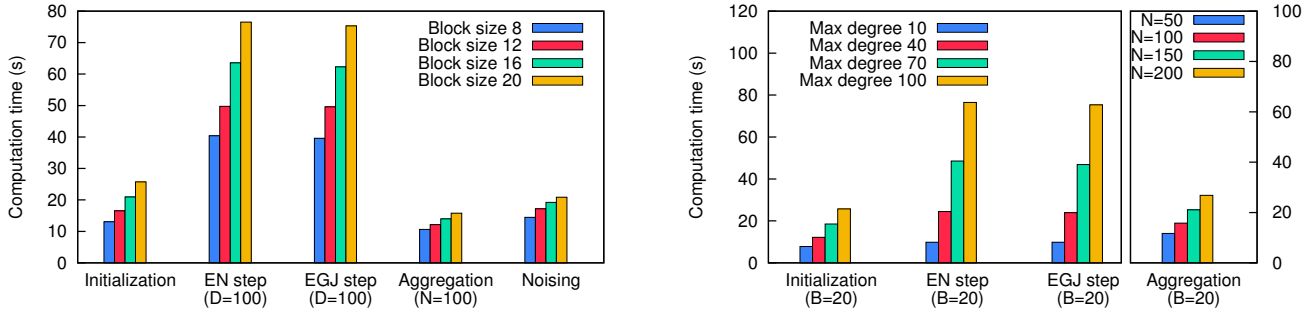

Figure 3: Computation time spent on MPC, with different block sizes (left), and different values for D and N (right).

time [59], which is based on an implementation of the GMW protocol [34] provided by Choi et al. [17]; 2) a distributed execution engine for graph algorithms; and 3) an implementation of the communication protocol from Section 3.5, based on the cryptographic primitives in the OpenSSL library. Our prototype generates Laplace noise using a circuit design from Dwork et al. [23]. To save computation and bandwidth, we apply a widely used ElGamal optimization [44] that reuses the same ephemeral key for each of the L bits in the share but requires L different public keys. Excluding Wysteria, our prototype consists of 11,904 lines of Java and 953 lines of C.

Unless otherwise noted, we conducted our experiments on Amazon EC2. We used up to 100 `m3.xlarge` instances, which each have four virtual Intel Xeon E5-2670 v2 2.5 GHz CPUs, 15 GB of memory, and two 40GB partitions of SSD-based storage. All the instances were located in the same EC2 region. For elliptic curves, we selected the NIST/SECG curve over a 384-bit prime field (`secp384r1`); this offers security equivalent to 192-bit symmetric cryptography, which is more than enough to defend against current cryptanalytic capabilities. We kept the default parameters for Wysteria and GMW: shares had a length of 12 bits (stored as 13 bytes), and the statistical security parameter for GMW was $k = 80$.

5.2 Microbenchmarks: Computation

DStress contains two main sources of computation cost: the MPC invocations that are used to perform the steps of the graph algorithm, and the cryptographic operations in the communication protocol. We evaluate each in turn.

MPC invocations: DStress performs four different kinds of operations in MPC: 1) the initialization step that generates the shares of each node's initial state; 2) the graph algorithm's computation step; 3) the graph algorithm's aggregation step; and 4) the final addition of Laplace noise. To quantify the cost of each, we performed a series of microbenchmarks in which we ran each MPC in isolation, using only Wysteria, for different block sizes. Since the computation steps in Eisenberg-Noe (EN) and Elliot-Golub-Jackson (EGJ) are different, we ran two separate experiments for this step.

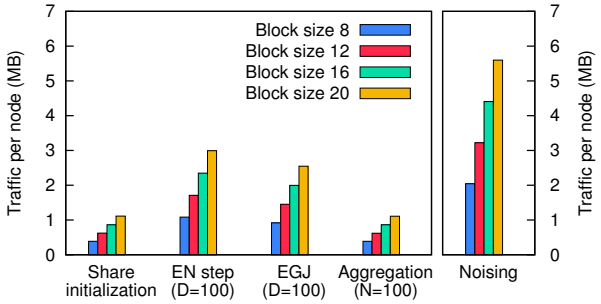

Figure 4: Per-node traffic generated by MPC computation steps with different block sizes.

The left part of Figure 3 shows the end-to-end completion times varied with the block size. There is a linear dependence, which is consistent with the theoretical complexity of GMW (the total cost is quadratic, but the nodes are working in parallel). We note that a block size of 20 is plausible in our setting: recall that the block size must be greater than the collusion bound k, and, to our knowledge, the largest known instance of collusion in the banking world was the LIBOR scandal, which involved 16 banks [60].

The right part of Figure 3 shows how the time for the initialization and computation steps varied with the degree bound D, and how the time for the aggregation step varied with the number of nodes N. Again, the dependencies are roughly linear; this is because the corresponding MPC circuits are fairly simple, so the number of gates depends mostly on the number of inputs.

Message transfers: To quantify the cost of the message transfer protocol from Section 3.5, we measured the time needed to transfer a single 12-bit message between two blocks of different sizes. We found that the end-to-end completion time was roughly proportional to k, from 285 ms with an 8-node block to 610 ms with a 20-node block. This is expected because each node in the block must encrypt $k+1$ subshares. There is a quadratic component as well because a single node must combine the $(k+1)^2$ encrypted subshares using the additive homomorphism, but this involves simple multiplications; the cost is dominated by the exponentiations, which are far more expensive.

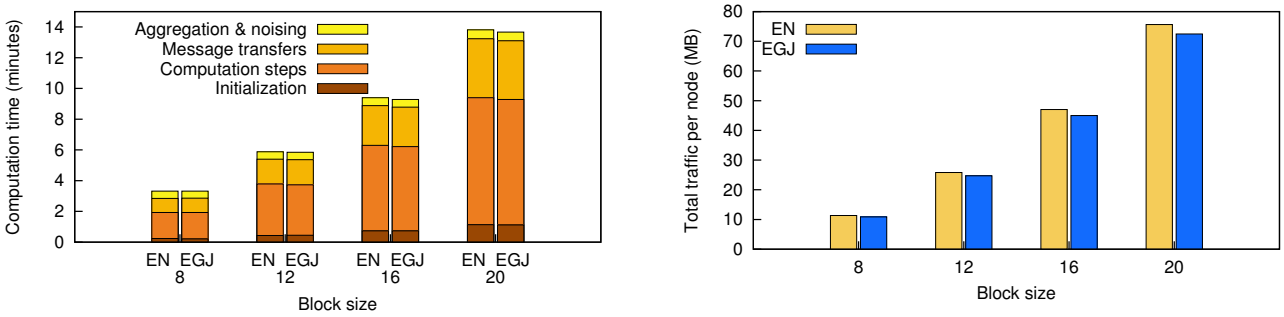

Figure 5: Computation time (left) and per-node traffic (right) for an end-to-end run on $N = 100$ vertexes (with maximum degree $D = 10$), while performing $I = 7$ iterations of Eisenberg-Noe (EN) and Elliot-Golub-Jackson (EGJ).

5.3 Microbenchmarks: Bandwidth

To quantify DStress's bandwidth cost, we measured the average amount of traffic that each node generated during the microbenchmarks from Section 5.2. As before, we examine the MPC and message transfer steps separately.

MPC invocations: Figure 4 shows our results for each of the five MPC circuits we have identified in Section 5.2. The traffic per node is roughly proportional to the block size $k+1$. Again, this is expected: although the total amount of traffic in GMW increases quadratically with the number of participants, the load is shared by $k+1$ nodes. We note that the absolute numbers are low and never exceed 6 MB per node, even for the comparatively large noising circuit. This is because Wysteria's GMW implementation includes oblivious transfer extensions [41, 46] as an optimization.

Message transfers: The amount of traffic for message transfers varies with the roles of the nodes. When the protocol is invoked for an edge (i, j), node i's load is the highest, since it receives $(k+1)^2$ encrypted subshares from B_i. In our experiments, this amounted to between 97 kB (with 8-node blocks) and 595 kB (with 20-node blocks). The nodes in B_i each send $k+1$ encrypted subshares, and j sends $k+1$ encrypted shares; thus, their traffic is linear in k and never exceeded 29 kB per node in our experiments. The nodes in B_j each receive a single encrypted share, regardless of the block size, so they handle a constant amount of traffic, about 1.4 kB.

Since we ran our experiments on EC2, neither propagation delays nor bandwidth constraints were major factors. This would be different in a wide-area deployment; however, since both MPC and the message transfers use relatively little bandwidth, we do not expect the network to become a major bottleneck in a wide-area setting.

5.4 End-to-end cost

To get a sense of the total cost of a DStress execution, we performed end-to-end runs with both EN and EGJ, using a synthetic graph with $N = 100$ banks, a degree limit of $D = 10$, and $I = 7$ iterations. As before, we varied the block size, and we measured the completion time and the average amount of traffic that was sent by each node.

Figure 5 shows our results. Although, as we have seen, the runtime of the individual operations is linear in k, the overall runtime varied roughly with $O(k^2)$; this is because, if we keep the number of nodes N constant while increasing k, each node must also participate in more blocks. (The actual dependence is not perfectly quadratic because each node handles multiple blocks in parallel, and the corresponding computations can be overlapped when one of them blocks.)

5.5 Scalability

In 2015, there were roughly 1,750 large commercial banks in the United States [54]. Due to our limited budget, we were unable to perform experiments with that many nodes; instead, we estimate the cost using results from our microbenchmarks.

Given values for the degree bound D, the number of nodes N, the collusion bound k, and the number of iterations I, it is easy to estimate the cost of the initialization, computation, and communication steps. For aggregation, we assume a two-level aggregation tree with degree 100 – that is, DStress would first aggregate the values from groups of 100 nodes (in parallel) and then further aggregate the results before the final noising step. We conservatively use a degree bound of $D = 100$ and a block size of $k+1 = 20$, and we assume that nodes cannot overlap computations from different blocks.

Obtaining realistic values for I is nontrivial because the exact structure of the banking network is not known, and cannot be fully inferred from the public (aggregate) disclosures. However, work in theoretical economics [18] has shown how to infer at least an approximate graph from public data. We reconstructed graphs based on this work, and found that $I = \log_2 N$ is enough to allow the algorithm to converge. We omit the details here, but they are available in our technical report [57].

Figure 6 shows our estimates for different network sizes; the red circles show the results from actual EC2 runs (N=20 and N=100) we performed for validation. (Recall that actual runs tend to be a bit faster than predicted because of the overlap between different block computations.) Based on these results, we estimate that an end-to-end run of Eisenberg-Noe for the entire U.S. banking system ($N = 1,750$, $D = 100$)

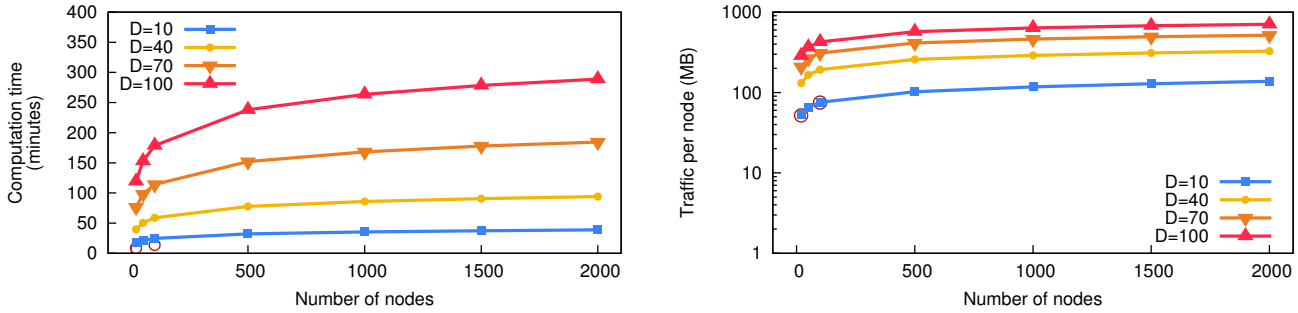

Figure 6: Projected computation cost (left) and per-node traffic (right) for end-to-end runs of EN on networks of different sizes. The red circles are validation points from two actual runs on EC2, with N=20 and N=100 nodes, respectively, and with D=10.

would take about 4.8 hours and consume about 750 MB of traffic. These costs seem low enough to be practical.

We did not compare DStress to prior work because we are not aware of any other system that efficiently offers guarantees that are comparable to those of DStress. However, one plausible baseline approach is to naïvely perform the entire computation as a single, monolithic MPC. If we ignore the details (such as the prorating, the comparisons, and the final matrix inversion), the closed form of an algorithm like EN essentially raises an $N \times N$ matrix to the I.th power. To estimate how long this would take, we wrote a simple Wysteria program that multiplies two square matrices, and we ran it for different values of N. As expected, the end-to-end completion time rose quickly, from 1.8 minutes for $N = 10$ to 40 minutes for $N = 25$. This is expected because the asymptotic complexity of matrix multiplication is $O(N^3)$. (Note that data-dependent optimizations cannot be applied because the data in the matrix is private.) We were unable to run the experiment for $N > 25$ because Wysteria ran out of memory, but we extrapolate that raising a 1750x1750 matrix to the $I - 1 = 11$th power would take $(1750/25)^3 * 40 * 11$ minutes, or about 287 years. This suggests that systemic risk detection using plain MPC would be infeasible in practice.

6. Related work

Differential privacy: There is a rich body of work on differentially private analytics for relational data [22], but there are much fewer results for graph data. [42] presents some private algorithms that offer *edge*-differential privacy, including $k-$triangle counting and $k-$star counting, but it is often difficult to give good accuracy with this approach because many algorithms have a high sensitivity to edge changes. Restricted sensitivity [10] takes advantage of the queriers' prior beliefs about the data to achieve higher accuracy. Our systemic-risk case study uses a slightly different guarantee, *dollar*-differential privacy, which was first proposed by Flood et al. [30, §4.3].

Distributed query processors: The first differentially private query processors, such as PINQ [49], Fuzz [38], and Airavat [61] assumed a centralized setting in which the analyst has access to all the private data. Later systems added support for distributed data, but they typically focus on a specific class of queries: for instance, PDDP [16] can build histograms, and DJoin [51] can process certain types of joins. Narayan et al. [52] sketched a system that can run iterative graph algorithms but offers weaker privacy guarantees than DStress – for instance, it leaks the some information about the structure of the graph. To our knowledge, DStress is the first practical system to support iterative graph algorithms with strong differential privacy guarantees.

Secure Multiparty Computation: Most practical MPC implementations are either based on the GMW protocol [34], which expresses computations as boolean circuits [7, 17], or based on the BGW protocol [8], which expresses computations as arithmetic circuits [15, 20, 65]. A direct comparison between protocols of the two main strands is not straightforward. In general, systems that use BGW, such as PICCO [65] or SEPIA [15], can offer better performance for applications that mostly use arithmetic operations. However, not all applications are of this type: for instance, Choi et. al. [17] showed that boolean-circuit systems outperform arithmetic-circuit systems for a specific class of matching algorithms. The best appropriate choice of MPC protocol for systemic risk algorithms is an open question; we selected GMW and [17] because both EGJ and EN can be expressed as graph computations, which seem to be a closer match to the algorithms described in [17]. In principle, our approach – breaking up a large MPC computation into smaller computations – should be applicable to the BGW protocol as well.

Recently, work on two-party secure computation (2PC) has started considering graph computations as well [55, 56]. Nayak et al. [55] identifies two key challenges in extending secure computation to graphs: one needs to protect the privacy of data as well as the graph topology. The solution described in [55] achieves the goals by obliviously sorting a combined list of all graph vertices and edges using garbled circuits. Unfortunately, full MPC is several orders of magnitude slower than 2PC; hence, this approach would face the same efficiency challenges that we detailed in 2.2.

We emphasize that we are not the first to consider the use of MPC for *differentially* private computations (see, e.g.,

[5, 23]). Our contribution is an efficient, scalable protocol for executing graph algorithms in a distributed setting without revealing the structure of the graph.

Message transfer protocol: Using ElGamal for its key randomization property has been considered in the literature before [21, 32, 37]. In fact, work concurrent to ours [21] presents an ElGamal-based construction which is similar to our message transfer protocol. Unfortunately, that solution is not additively homomorphic and cannot be directly used in DStress.

7. Conclusion

In this paper, we have presented DStress, a system that can efficiently analyze large, distributed graphs with confidential information. DStress's programming model resembles that of other frameworks for graph analytics; however, DStress executes programs in a distributed fashion, using a combination of secret sharing, small multi-party computations, and a special protocol for transferring messages without revealing the structure of the graph. As a result, DStress only needs a few hours to run computations with hundreds of participants, whereas a naïve application of multi-party computation would take many years.

We have also studied one concrete use case of DStress that we have taken from the economics literature: the computation of systemic risk in financial networks. We have shown that DStress can implement two state-of-the-art models of systemic risk; our experimental results suggest that these models could be evaluated on all the large commercial banks in the United States within about five hours, using only one commodity machine at each bank.

Acknowledgments

We thank our shepherd, Raluca Popa, and the anonymous reviewers for their thoughtful comments and suggestions. Benjamin Pierce, Justin Hsu, Brett Hemenway, Nishanth Chandran, Frank McSherry, and Aaron Roth provided helpful feedback on earlier versions of this paper. We are grateful to Aseem Rastogi, Matthew Hammer, and Michael Hicks for their extensive support in using the Wysteria circuit compiler and MPC implementation. This work was supported in part by NSF grants CNS-1054229 and CNS-1513694, as well as the Intel-NSF Partnership for Cyber-Physical Systems Security and Privacy (CNS-1505799).

References

[1] E. A. Abbe, A. E. Khandani, and A. W. Lo. Privacy-preserving methods for sharing financial risk exposures. *The American Economic Review*, 102(3):65–70, 2012.

[2] Bank for International Settlements. International regulatory framework for banks (Basel III) website. `http://www.bis.org/bcbs/basel3.htm`.

[3] M. Barbaro, T. Zeller, and S. Hansell. A face is exposed for AOL searcher No. 4417749. *New York Times*, Aug 9, 2006.

[4] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. In *Proc. POPL*, 2013.

[5] A. Beimel, K. Nissim, and E. Omri. Distributed private data analysis: Simultaneously solving how and what. In *Proc. CRYPTO*, 2008.

[6] R. M. Bell and Y. Koren. Lessons from the Netflix prize challenge. *ACM SIGKDD Explorations Newsletter*, 9(2):75–79, 2007.

[7] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A system for secure multi-party computation. In *Proc. ACM CCS*, 2008.

[8] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proc. STOC*, 1988.

[9] D. Bisias, M. Flood, A. W. Lo, and S. Valavanis. A survey of systemic risk analytics. *Annu. Rev. Financ. Econ.*, 4(1):255–296, 2012.

[10] J. Blocki, A. Blum, A. Datta, and O. Sheffet. Differentially private data analysis of social networks via restricted sensitivity. In *Proc. ITCS*, 2013.

[11] Board of Governors of the Federal Reserve System. Dodd-Frank Act Stress Test 2015: Supervisory stress test methodology and results, Mar. 2015. `http://www.federalreserve.gov/newsevents/press/bcreg/bcreg20150305a1.pdf`.

[12] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Proc. ESORICS*, 2008.

[13] R. Bookstaber, J. Cetina, G. Feldberg, M. Flood, and P. Glasserman. Stress tests to promote financial stability: Assessing progress and looking to the future. *Journal of Risk Management in Financial Institutions*, 7(1):16–25, 2014.

[14] S. P. Borgatti, A. Mehra, D. J. Brass, and G. Labianca. Network analysis in the social sciences. *Science*, 323(5916):892–895, 2009.

[15] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *Proc. USENIX Security*, 2010.

[16] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards statistical queries over distributed private user data. In *Proc. NSDI*, 2012.

[17] S. G. Choi, K.-W. Hwang, J. Katz, T. Malkin, and D. Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. In *Proc. CT-RSA*. Springer, 2012.

[18] J. F. Cocco, F. J. Gomes, and N. C. Martins. Lending relationships in the interbank market. *Journal of Financial Intermediation*, 18(1):24–48, 2009.

[19] R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. *Transactions on Emerging Telecommunications Technologies*, 8(5):481–490, 1997.

[20] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *PKC*, 2009.

[21] Y. Dodis, I. Mironov, and N. Stephens-Davidowitz. Message transmission with reverse firewalls—secure communication on corrupted machines. Technical report, Cryptology ePrint Archive, Report 2015/548, 2015. http://eprint. iacr. org/2015/548, 2015.

[22] C. Dwork. Differential privacy: A survey of results. In *Proc. TAMC*, 2008.

[23] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. In *Proc. EUROCRYPT*, 2006.

[24] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference (TCC)*, 2006.

[25] L. Eisenberg and T. H. Noe. Systemic risk in financial systems. *Management Science*, 47(2):236–249, 2001.

[26] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[27] M. Elliott, B. Golub, and M. O. Jackson. Financial networks and contagion. *The American economic review*, 104(10):3115–3153, 2014.

[28] J. Fantuzzo and D. P. Culhane. *Actionable intelligence: Using integrated data systems to achieve a more effective, efficient, and ethical government*. Palgrave Macmillan, 2015.

[29] Federal Deposit Insurance Corporation (FDIC). Dodd-Frank Act Stress Test. February 17, 2017; `https://www.fdic.gov/regulations/reform/dfast/index.html`.

[30] M. Flood, J. Katz, S. Ong, and A. Smith. Cryptography and the economics of supervisory information: Balancing transparency and confidentiality. *U.S. Department of Treasury Office of Financial Research Working Paper Series*, 1(11), 2013.

[31] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *Proc. POPL*, 2013.

[32] C. Gentry, S. Halevi, and V. Vaikuntanathan. i-hop homomorphic encryption and rerandomizable yao circuits. In *CRYPTO*, 2010.

[33] A. Ghosh, T. Roughgarden, and M. Sundararajan. Universally utility-maximizing privacy mechanisms. *SIAM Journal on Computing*, 41(6):1673–1693, 2012.

[34] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proc. STOC*, 1987.

[35] J. E. Gonzalez, Y. Low, H. Gu, D. Bickso, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. OSDI*, 2012.

[36] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proc. OSDI*, 2014.

[37] J. Groth. Rerandomizable and replayable adaptive chosen ciphertext attack secure cryptosystems. In *Theory of Cryptography*, 2004.

[38] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proc. USENIX Security*, 2011.

[39] B. Hemenway and S. Khanna. Sensitivity and computational complexity in financial networks. In submission; manuscript available from `http://www.cis.upenn.edu/~sanjeev/papers/financial_network.pdf`, 2015.

[40] J. Hsu, M. Gaboardi, A. Haeberlen, S. Khanna, A. Narayan, B. C. Pierce, and A. Roth. Differential privacy: An economic method for choosing epsilon. In *Proc. IEEE CSF*, 2014.

[41] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Proc. CRYPTO*, 2003.

[42] V. Karwa, S. Raskhodnikova, A. Smith, and G. Yaroslavtsev. Private analysis of graph structure. *Proc. VLDB Endowment*, 4(11):1146–1157, 2011.

[43] V. E. Krebs. Mapping networks of terrorist cells. *Connections*, 24(3):43–52, 2002.

[44] K. Kurosawa. Multi-recipient public-key encryption with shortened ciphertext. In *PKC*, 2002.

[45] D. Lazer, A. Pentland, L. Adamic, S. Aral, A.-L. Barabási, D. Brewer, N. Christakis, N. Contractor, J. Fowler, M. Gutmann, T. Jebara, G. King, M. Macy, D. Roy, and M. Van Alstyne. Computational social science. *Science*, 323(5915):721–723, 2009.

[46] B. Li, H. Li, G. Xu, and H. Xu. Efficient reduction of 1 out of n oblivious transfers in random oracle model. *IACR Cryptology ePrint Archive*, 2005:279, 2005.

[47] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *Proc. UAI*, 2010.

[48] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proc. SIGMOD*, 2010.

[49] F. McSherry. Privacy integrated queries. In *Proc. SIGMOD*, 2009.

[50] I. Mironov, O. Pandey, O. Reingold, and S. Vadhan. Computational differential privacy. In *Proc. CRYPTO*, 2009.

[51] A. Narayan and A. Haeberlen. DJoin: Differentially private join queries over distributed databases. In *Proc. OSDI*, 2012.

[52] A. Narayan, A. Papadimitriou, and A. Haeberlen. Compute globally, act locally: Protecting federated systems from systemic threats. In *Proc. HotDep*, 2014.

[53] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Proc. IEEE S&P*, 2008.

[54] National Information Center of the Federal Reserve System. Insured U.S. chartered commercial banks that have consolidated assets of $300 million or more. `http://www.federalreserve.gov/releases/lbr/current/default.htm`, 2014.

[55] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: Parallel secure computation made easy. In *Proc. IEEE S&P*, 2015.

[56] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. Privacy-preserving matrix factorization. In *Proc. ACM CCS*, 2013.

[57] A. Papadimitriou, A. Narayan, and A. Haeberlen. DStress: Efficient differentially private computations on distributed data. Technical Report MS-CIS-17-03, University of Pennsylvania, 2017.

[58] D. Proserpio, S. Goldberg, and F. McSherry. Calibrating data to sensitivity in private data analysis: A platform for differentially-private analysis of weighted datasets. In *Proc. VLDB*, 2014.

[59] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *Proc. IEEE S&P*, 2014.

[60] N. Raymond and A. Viswanatha. U.S. regulator sues 16 banks for rigging Libor rate. Reuters, March 14, 2014; `http://www.reuters.com/article/2014/03/14/us-fdic-libor-idUSBREA2D1KR20140314`.

[61] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *Proc. NSDI*, 2010.

[62] M. K. Sparrow. The application of network analysis to criminal intelligence: An assessment of the prospects. *Social networks*, 13(3):251–274, 1991.

[63] A. Yao. Protocols for secure computations. In *Proc. FOCS*, 1982.

[64] E. Zhai, R. Chen, D. I. Wolinsky, and B. Ford. Heading off correlated failures through independence-as-a-service. In *Proc. OSDI*, 2014.

[65] Y. Zhang, A. Steele, and M. Blanton. Picco: A general-purpose compiler for private distributed computation. In *Proc. ACM CCS*, 2013.