Differentially-Private Network Trace Analysis

Frank McSherry and Ratul Mahajan
Microsoft Research

Overview

.

Overview

Question: Is it possible to conduct network trace analyses in a way that provides strict formal "differential privacy" guarantees?

Methodology: Select a representative sample of network trace analyses from the literature, reproduce with differential privacy.

Results: We were able to reproduce every analysis we attempted. The privacy/accuracy trade-off varied by analysis; caveats hold.

Toolkit and analyses we wrote are available at:

`http://research.microsoft.com/pinq/networking.aspx`

Network Trace Analysis

Much of networking research relies on access to good, rich data. Network traces (long lists of observed packets) are one example.

The research process is complicated by a tension between:

Utility: The trace should reflect actual network behavior.
Privacy: The trace could reflect actual network behavior.

While this looks irreconcilable, there is an important difference.

Utility requirements are typically for aggregate statistics.
Privacy requirements are typically for individual behavior.

Not obviously hopeless. But, how to proceed?

Privacy in NTA: Related Work

We aren't the first people to look at privacy in trace analysis. Not going to be the last, either.

Some examples of other approaches:

Trace anonymization: Sometimes it works, sometimes it doesn't. Prefix-preserving anonymization is a good example of challenge.

Code to Data: Data unmolested, but code may be inscrutable. Current proposals either seem to rely on experts (*eg* SC2D, trol) or leak [bounded amounts of] arbitrary information (Mittal et al).

Secure Multi-party Computation: Same as for Code to Data.

Our aim: Formal guarantees first. As useful as possible next.

Differential Privacy

Differential privacy formally constrains computations to conceal the presence or absence of individual records:

Definition: A randomized M gives **ϵ-differential privacy** iff: for all input datasets A, B and any possible output S,

$$\Pr[M(A) = S] \;\leq\; \Pr[M(B) = S] \times \exp(\epsilon \times |A \ominus B|) \; .$$

Ensures: Any event S "equally likely" with/without your data.

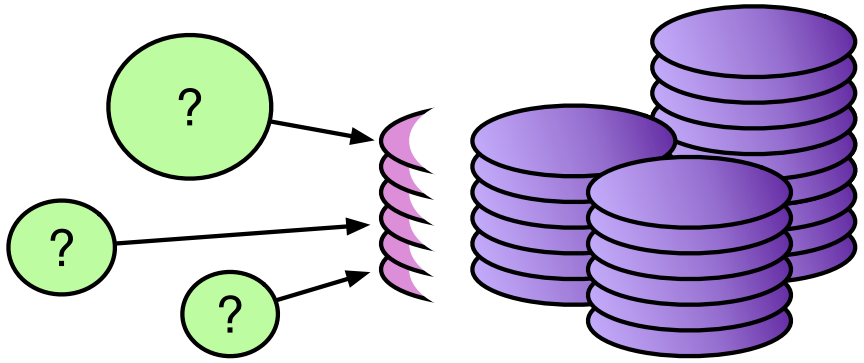
1. Doesn't prevent disclosure. Ensures disclosure not our fault.
2. No computational / informational assumptions of attackers.
3. Agnostic to record type. Could be PII, binary data, anything.

Simplest example of DP computation is Count + Noise.

Privacy Integrated Queries

PINQ: Common platform for differentially-private data analyses.

1. Provides interface to data that looks very much like LINQ.
2. All access through the interface gives **differential privacy**.



Analysts write arbitrary LINQ code against data sets, using C#.
No privacy expertise needed to produce analyses. (but it helps)

We are going to try to write Network Trace Analyses using PINQ.

What's the Hard Part?

While DP has many great features, it comes with challenges too:

Some we will deal with here:

1. Achieving DP involves perturbing answers to queries (noise).
   A: Reframe analyses using statistically robust measurements.

2. Programming in PINQ requires high-level, declarative queries.
   A: This can certainly require some creativity/reinterpretation.

Some are still challenges, and should be discussed (none fatal):

3. Masking just a few packets does not mask a "person".

4. The guarantees degrade the more a dataset is "used".

5. ... more ...

Worm Fingerprinting in LINQ

One view of a worm (from Singh et al) is as a payload seen destined for many distinct source and destination IP addresses.

```
var trace = LoadTrace(); // type can be as simple as Packet[]

var worms = trace.GroupBy(pkt => pkt.Payload)
                 .Where(group => group.Select(pkt => pkt.SrcIP)
                                      .Distinct()
                                      .Count() > srcThreshold)
                 .Where(group => group.Select(pkt => pkt.DstIP)
                                      .Distinct()
                                      .Count() > dstThreshold);

Console.WriteLine(worms.Count());
```

Identifies worms and then reports their number.

Worm Fingerprinting in PINQ

One view of a worm (from Singh et al) is as a payload seen destined for many distinct source and destination IP addresses.

```
var trace = LoadTrace(); // type is now PINQueryable<Packet>

var worms = trace.GroupBy(pkt => pkt.Payload)
                 .Where(group => group.Select(pkt => pkt.SrcIP)
                                      .Distinct()
                                      .Count() > srcThreshold)
                 .Where(group => group.Select(pkt => pkt.DstIP)
                                      .Distinct()
                                      .Count() > dstThreshold);

Console.WriteLine(worms.Count(epsilon));
```

Identifies worms and then reports their number, approximately.

Building Analysis Tools

At this point, we can start to build useful tools in PINQ.
For example: Cumulative Density Functions. (Approach 1/3)

```
IEnumerable<double> CDF(PINQueryable<int> input, int maximum, double epsilon)
{
   foreach (var entry in Enumerable.Range(0, maximum))
      yield return input.Where(x => x < entry)
                        .Count(epsilon / maximum);
}
```

Building Analysis Tools

At this point, we can start to build useful tools in PINQ.
For example: Cumulative Density Functions. (Approach 2/3)

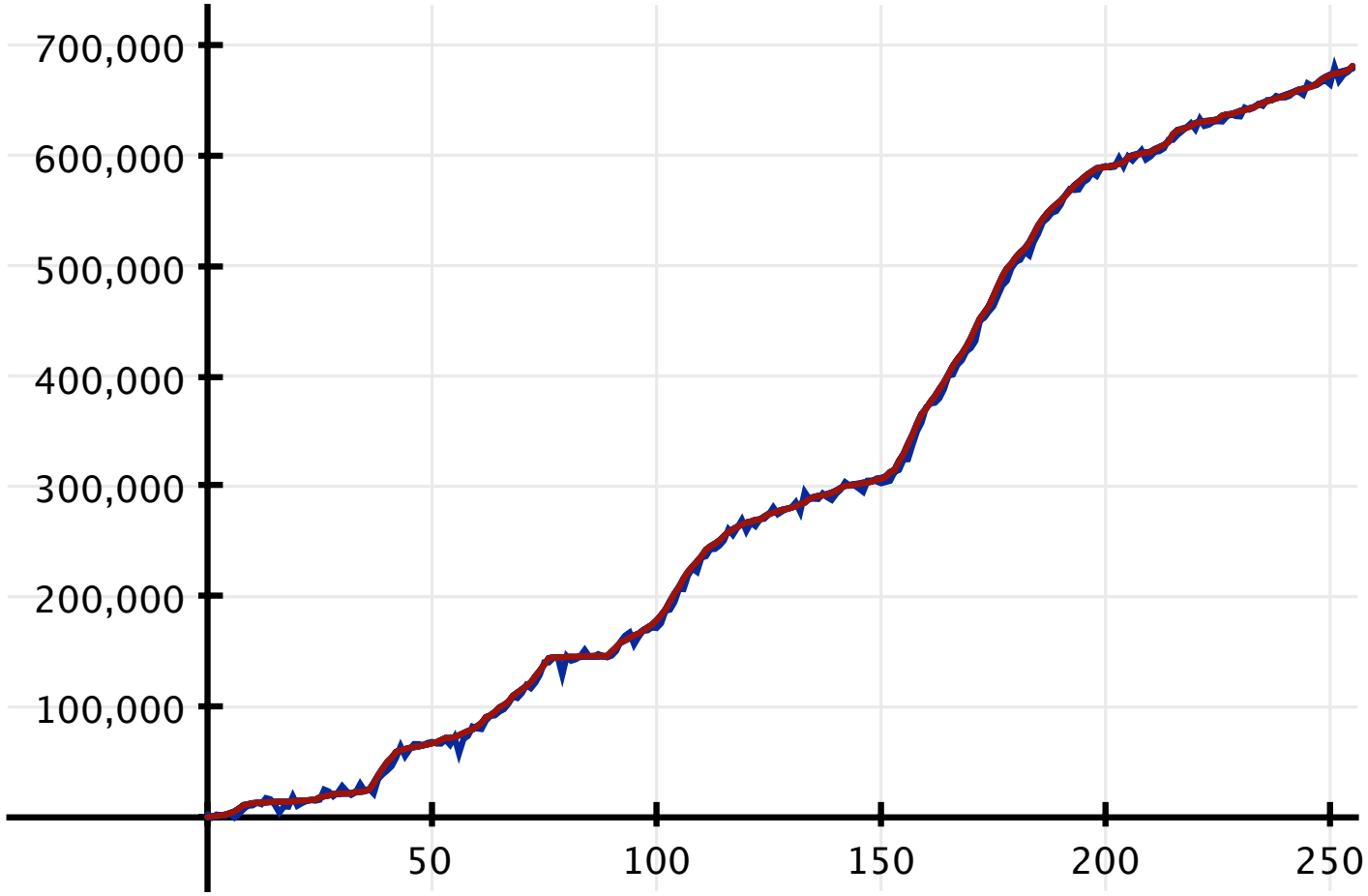
```
IEnumerable<double> CDF(PINQueryable<int> input, int maximum, double epsilon)
{
   var tally = 0;
   var parts = input.Partition(Enumerable.Range(0, maximum), x => x);
   foreach (var entry in Enumerable.Range(0, maximum))
   {
      tally = tally + parts[entry].Count(epsilon);
      yield return tally;
   }
}
```

Building Analysis Tools

At this point, we can start to build useful tools in PINQ.
For example: Cumulative Density Functions. (Approach 3/3)

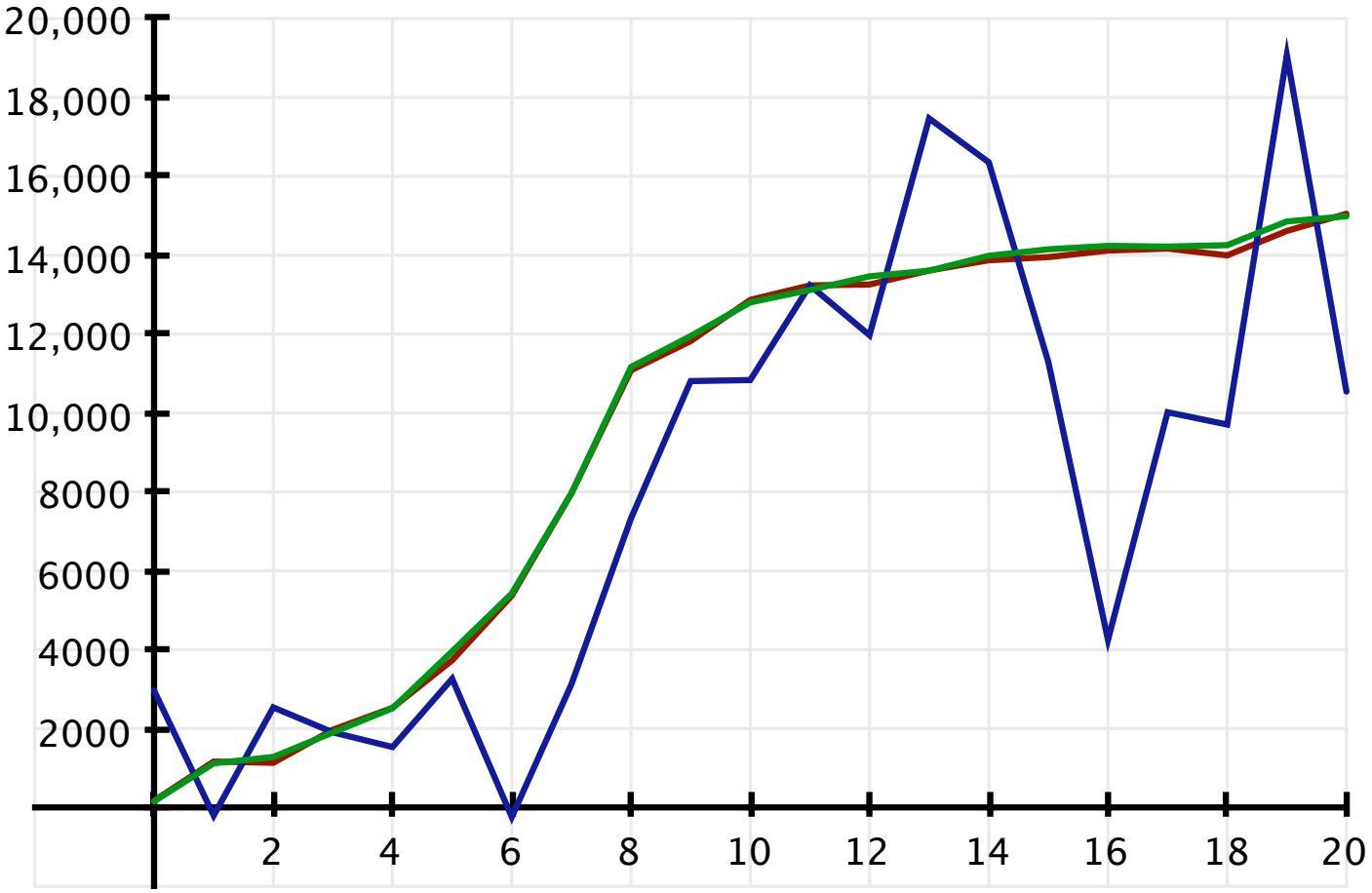
```
IEnumerable<double> CDF(PINQueryable<int> input, int maximum, double epsilon)
{
   if (maximum == 0)
      yield return input.Count(epsilon);

   else
   {
      var parts = input.Partition(new int[] { 0, 1 }, x => x / (maximum / 2));
      foreach (var count in CDF(parts[0], maximum / 2, epsilon)
         yield return count;

      var cache = parts[0].Count(epsilon);

      parts[1] = parts[1].Select(x => x - maximum / 2);
      foreach (var count in CDF(parts[1], maximum / 2, epsilon)
         yield return count + cache;
   }
}
```

Example: CDFs, eps = 0.1



Blue = CDF1, Green = CDF2, Red = CDF3

Example: CDFs, eps = 0.1



Blue = CDF1, Green = CDF2, Red = CDF3

Another Tool: Strings

Given a collection of strings, list the frequently occurring strings.
Sounds like bad privacy, but +/- one record is still hidden.

```
// enumerates frequently occurring strings in input starting with prefix
IEnumerable<string> Strings(PINQueryable<string> input, string prefix)
{
   // split input into those equal to prefix, and those that are prefixes
   var exact = input.Partition(new bool[] {true, false}, x => x == prefix);

   // if we have enough records equal to prefix, return it
   if (exact[true].Count(epsilon) > confidence / epsilon)
      yield return prefix;

   // other records contribute to each possible extension of prefix
   var parts = exact[false].Partition(keys, x => x[prefix.Length]);
   foreach (var key in keys)
      if (parts[key].Count(epsilon) > confidence / epsilon)
         foreach (var result in Strings(parts[key], prefix + key))
            yield return result;
}
```

Example: Strings, eps = 0.1

Finding frequent hex strings in (hashes of) packet payloads:

```
Strings(trace.Select(packet => packet.Payload), "", 0.1);
```

Top 10 payload recovered, in order, with relatively small error.

hash(payload)	true count	est. count	% err
2D2816FECDCAB780	3038504	3038500.005	-0.000
F389B84545A38BAF	92494	92505.050	0.012
E41903DCF7D86F2F	41600	41606.893	0.017
6F7E03DC833D6F2F	40279	40287.970	0.022
CD4F03DCE10E6F2F	40084	40087.437	0.009
B68503DCCA446F2F	37431	37448.584	0.047
58B403DC6C736F2F	36526	36537.877	0.033
41EA03DC55A96F2F	29625	29624.397	-0.002
9FBB03DCB37A6F2F	20715	20711.169	-0.018
7EEEB845D1088BAF	18976	18980.823	0.025

Worm Fingerprinting: Redux

Actually enumerating payloads with significant src/dst counts:

```
// enumerates actual payloads with high src/dst dispersal
IEnumerable<string> FindWorms(PINQueryable<Packet> trace)
{
   var loads = Strings(trace.Select(packet => packet.Payload), "");

   var parts = trace.Partition(loads, packet => packet.Payload);

   foreach (var load in loads)
   {
      var srcCount = parts[load].Select(packet => packet.SrcIP)
                                .Distinct()
                                .Count(epsilon);

      var dstCount = parts[load].Select(packet => packet.DstIP)
                                .Distinct()
                                .Count(epsilon);

      if (srcCount > srcThreshold && dstCount > dstThreshold)
         yield return load + " " + srcCount + " " + dstCount;
   }
}
```

Other Tools and Analyses

We wrote a few other tools and analyses as well.

Tools: CDFs, Frequent Strings, Frequent Itemset Mining.

Analyses: Several types of trace analyses, varying granularities.
Two Qs: can we write the analysis faithfully, and is it accurate?

Packet-level analyses	Expressibility	High accuracy
Packet size and port dist.	faithful	strong privacy
Worm fingerprinting	faithful	weak privacy

Flow-level analyses		
Common flow properties	connections vs. flows	strong privacy
Stepping stone detection	windows approximated	medium privacy

Graph-level analyses		
Anomaly detection	faithful	strong privacy
TTL Clustering	used k-means	weak privacy

Wrapping Up

Studying use of differential privacy for network trace analysis. Seems promising. Formal privacy guarantees and good results.

`http://research.microsoft.com/pinq/networking.aspx`

Many open questions to tackle, discuss, follow up on:

1. Are DP guarantees for packets good enough / meaningful? Can the traces / datasets be massaged to make them so?

2. Can we conduct research, rather than reproduce research?

3. Design analyses with PINQ in mind, improve priv/accuracy? Many "expressibility" failures weren't much different in spirit.

4. Can we augment PINQ with network-specific functionality? Segmenting flows into connections would have helped, works.