Verifiable Distributed Aggregation Functions
draft-patton-cfrg-vdaf-01

Presented at IETF 113 (CFRG)
Authors: Christopher Patton (speaker), Richard Barnes, Phillipp Schoopmann
Context

- PPM working group's objective is to standardize multi-party computation for privacy preserving measurement
  - draft-gpew-priv-ppm-01 – the "PPM protocol", candidate for PPM working group adoption
    - Specifies end-to-end verification and aggregation of measurements over HTTPS
  - This document (draft-patton-cfrg-vdaf-01) – The core cryptographic component of the PPM protocol

- Ask for the CFRG: Is draft-patton-cfrg-vdaf-01 ready for adoption by the working group?
Overview of VDAFs

1. **Shard** – Each Client splits its *measurement* into *input shares* and sends one share to each Aggregator.

2. **Prepare** – Aggregators prepare each set of input shares for aggregation. Input shares are mapped to *output shares* using an optional *aggregation parameter*.

3. **Aggregate** – Each Aggregator combines its output shares into an *aggregate share* and sends it to the Collector.

4. **Unshard** – Collector combines aggregate shares into the aggregate result.
Candidate Constructions – Prio [CGB17, BBCG+19]

1. **Shard** – Client i splits $m[i]$ into secret shared vectors $X[i]$ and $X[i]$ over some finite field.

2. **Prepare** – For each i, Aggregators interact among themselves in order to verify that $X[i] + X[i]$ is a valid input.

3. **Aggregate** –

4. **Unshard** – Collector computes $a = Y + Y$. 
Candidate Constructions – Poplar [BBCG+21]

- Problem: securely compute the heavy hitters
  - Measurements: N-bit strings
    - \( N=3 \): \( 000, 111, 010, 011, 100, 110, 100, 010, 010, 101, 001 \)
  - Aggregate result: Strings with at least \( T \) hits
    - \( T=2 \): \( 010, 100 \)

- Solution: Incremental Distributed Point Functions (IDFPs)
  - Clients split their measurement into IDPF shares
  - Aggregators query IDPF shares on candidate prefixes
    - Input = \( 011 \): is 0 a prefix? Yes; is 1 a prefix? No; is 01 a prefix? Yes; …
  - Each aggregator holds a share of each query output.
    - Output shares are aggregatable into hit counts, i.e., the frequency of each candidate prefix.
Candidate Constructions – Poplar [BBCG+21]

1. **Shard** – Client generates IDPF shares from its input string.

2. **Prepare** – Each Aggregator queries its IDPF share at each candidate prefix. Aggregators interact in order to verify the output shares are well-formed (without revealing the output).

3. **Aggregate** – Each Aggregator combines its output shares into a share of the hit count for each candidate prefix.

4. **Unshard** – Collector combines hit count shares to get hit counts of each candidate prefix. *(Hit counts used to compute the next set of candidate prefixes.)*
Progress since IETF 112

- Minor syntax improvements
- Complete spec of Prio, including a reference implementation for generating test vectors
  - Reference implementation (Sage): https://github.com/cjpatton/vdaf/tree/main/poc
  - Rust implementation: https://docs.rs/prio/0.7.0/prio/vdaf/prio3
    - Lots of room for optimization, but fast enough for now

<table>
<thead>
<tr>
<th>VDAF</th>
<th>Client runtime</th>
<th>Client communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prio3Aes128Count</td>
<td>11.3 us</td>
<td>80 bytes</td>
</tr>
<tr>
<td>Prio3Aes128Histogram (10 buckets)</td>
<td>22.6 us</td>
<td>768 bytes</td>
</tr>
<tr>
<td>Prio3Aes128Sum (32 bits)</td>
<td>47.2 us</td>
<td>2,656 bytes</td>
</tr>
</tbody>
</table>
Next Steps

- Complete spec of **Poplar**
  - Incomplete implementations exist, none are interoperable.
    - C++: [https://github.com/google/distributed_point_functions](https://github.com/google/distributed_point_functions)
    - Rust: [https://docs.rs/prio/0.7.0/prio/vdaf/poplar1](https://docs.rs/prio/0.7.0/prio/vdaf/poplar1)
- Security analysis and fleshed out security considerations
- More VDAFs! Either in this document or elsewhere
- Enumeration of open issues: [https://github.com/cjpatton/vdaf/issues](https://github.com/cjpatton/vdaf/issues)
References