Securing Distributed Computation via Trusted Quorums

Yan Michalevsky, Valeria Nikolaenko, Dan Boneh
Setting

- Distributed computation over data contributed by users
- Communication through a central party
  - Untrusted/Malicious central party
  - Denial of access to raw data due to regulation
- Computations similar to map-reduce
  - Corrupted/colluding reducers
Coordinated distributed computation

Problematic to scale!
Coordinated distributed computation with groups
Applications

• Privacy preserving aggregation
  • Users report values and only the sum is revealed to the collector
• Log-chaining
  • Signed log records verified by consensus
• Cryptocurrencies
  • Users post transactions verified by consensus
• Computation of HUGE Merkle-trees and Merkle-signatures
Malicious server and corrupted workers
Aggregating reports

- “Do you have …”
- “... this app installed”
- “... this extension enabled”
- “... this type of malware”
- “... these medical conditions”
- etc.
Simplest Approach

Don't want aggregator to learn the values

\[ b_1, b_2, \ldots, b_N \]
More concretely

- Aggregator wants to know statistics
- Aggregator wants to preserve users privacy
Aggregator obliviousness (AO)

A protocol is *aggregator oblivious* if, at the end of the execution, the *aggregator leans nothing* about the reports *except* for their *sum*. If a portion of the reports is compromised (by collusion or attacks), the aggregator learns nothing else about the reports of the rest of the participants.
Requirements

• Single central party – the aggregator
• No p2p
  • All communication through the aggregator
  • PKI available
• Fault tolerance
  • Clients are not expected to be responsive in all rounds
• Reporters are not anonymous but reported values protected
Trusted party

\[ s = \sum b_i \]

Too strong: “Trusted party honest and not curious”
"Trusted party" is trusted not to collude with Google

Let $b_1, b_2, \ldots, b_N$ be the bit strings provided by trusted parties. Trusted party computes $ct = Enc_{pk}(\sum b_i)$ and sends it to Google. Google decrypts $ct$ with $sk$ to obtain $\sum b_i$. The trusted party is trusted not to collude with Google.
MASKING via secret shares

$r_1 + r_2 + \cdots + r_n + s = 0$

\[ s + \sum c_i = \sum v_i \]
Num. of rounds

- Protocol has to consist of at least 2 steps
- Otherwise, aggregator can perform a computation twice, second time excluding one of the reports
Threshold-homomorphic encryption

Homomorphic encryption

\[ Enc(m_1) \cdot Enc(m_2) = Enc(m_1 + m_2) \]

Threshold encryption

Secret-key sk is shared among N parties
any subset of K parties can decrypt

Several prior works use this method:
• Rastogi and Nath
• Jawurek and Kerschbaum
• And more...
1. Reporter $i$ encrypts $b_i$
   
   $$ct_i = Enc_{pk}(b_i)$$

2. $ct_i$ and proof of authenticity are sent to the aggregator

3. Reporter sends a ZKP that the encrypted value $b_i$ is 0 or 1
Protocol

4. Server picks groups of workers and sends them disjoint report sets
5. Worker verifies reports and computes the sum

Algorithm 1 VerifyReport

```plaintext
if not Verify_{pk,CA} (pk_i, Cert_i) then
    Return Fail
end if
if not Verify_{pk_i} (c_i || P_i || t_i || pk_i || Cert_i, σ_i) then
    Return Fail
else
    Return Success
end if
```
Protocol

6. Each worker computes the signature
\[ \sigma_j = \text{Sign}_{sk_{wj}} \left( \Sigma ||\text{count}||H_1 \right) \]
and sends back a tuple containing the sum, count, reports set hash \( H_1 \) and the signature
7. Each worker gets a tuple from the last iteration (sum, count, set)

```
Algorithm 2 VerifyPartialSum
validSigCounter ← 0
for all j ∈ 1..num. of signatures do
    if Verify_{pk_w_j} (Σ||count||H_1, \sigma_j) then
        validSigCounter ← validSigCounter + 1
    end if
end for
if validSigCounter ≥ SigMin then
    Return Success
else
    Return Fail
end if
```
Protocol

8. Each worker gets a tuple from the last iteration (sum, count, set)

9. If count is big enough, send back the decryption share of the sum

10. Aggregator reconstructs plaintext from shares
Threats

A malicious aggregator can
• Drop, insert or change reports
• Corrupt a portion of the workers
• Manipulate workers grouping to obtain majority
• Modify messages propagated to next round
Vulnerabilities

1. Aggregator can pick a group of colluding workers
2. Obtains a valid certificate for an arbitrary value
3. Can recover individual values or reduce the anonymity set
Honest Quorums

- Need to guarantee honest quorums in worker groups
- If the minority of corrupted workers is distributed uniformly across groups we have honest majority in each worker group
Fixed worker-group assignment

• Assignment of workers to groups is fixed (by e.g. hashing the public key of the worker)
• Incentivizes to corrupt a particular group (compulsion attacks)
Randomized assignment: Take 1

• Given a set of reports $c_1, c_2...c_M$ compute
  
  $H_1 = H(c_1 | c_2 | ... | c_M)$
  $H_2 = H(H_1)$

• Groups of workers derived from $H_2$
• Each worker can verify it is supposed to handle the reports it received by computing $H_1, H_2$ on its own

• Worker includes $H_1$ in certificate

• In the next iteration a worker verifies that for each partial sum it has signatures from the right workers
Randomized assignment: Take 1

PROBLEM

By substitution of one element in a report subset, the aggregator sends to send *roughly* the same content to multiple groups – reduces anonymity set!
Randomized assignment: Take 2

- Each reporter picks random $r_i$ and includes it in the report
- Aggregator builds Merkle-tree
- Grouping leaves by $H_1(r_i)$

\[
s_1 = H_2(r_1|r_3|r_1) \\
s_2 = H_2(r_4|r_5) \\
\vdots \\
H_1(s_1) = H_1(s_4) \\
H_1(s_2) = H_1(s_3)
\]
Randomized assignment: Take 2

• $r_{root}$ is random - serves to form worker groups:
  \[ H_3(x) = Hash(x \parallel r_{root}) \]

• Announces $r_{root}$ to workers

• $H_3(r_i) \mod N : (N - \#groups)$ determines group assignment for report
Randomized assignment verification

- Each worker verifies that it should process the received reports
- Computes $s_i$ values independently and includes in partial sum certificate
- At the last round workers verify that they obtained the expected $r_{root}$
Security

• **Claim:** our quorum forming scheme is secure in the random-oracle model

• Randomness of $r_{\text{root}}$ is by inductively applying random oracles to each stage of computing the Merkle-tree

• Inability to cheat the workers (verifiers) stems from collision resistance of a Merkle-hash tree

An extreme case where the aggregator controls a whole sub-tree
Attacks?

• Caveat: server can use a colluding reporter to generate reports with different $r_i$ values such that each one maps to a different worker group

• Won’t be discovered by workers in this case

• Low success probability for attack

• Birthday: probability of having 2 reports assigned to the same group

$$p(N, G) \approx 1 - \exp \left( -\frac{N^2}{2G} \right)$$

$$N = 1000, G = 1000$$
Implementation

- Java – high-level, conveniently parallelizable, fast and portable
- Entities decoupled by design – easy to deploy in a distributed setting using RMI or Hadoop map-reduce
- RSA signatures
  - Fast to verify
- Elgamal threshold encryption over the integers (need to switch to EC)
- Random oracles substituted by SHA-256
(Preliminary) Evaluation

- Amazon EC2 instance with 32 cores
- 10,000,000 reports in 19 minutes
  - Not really parallel
  - 10,000 workers
  - Up to 1000 worker groups
- Threshold-decryption takes a big chunk of the computation
Future work

- Evaluate in a more distributed setting
  - Profile different stages of the protocol
- Switch to elliptic curves
  - Smaller keys and better performance
- Sig. aggregation / batch verification of signatures
- Support computation of more aggregate functions
  - Min, max, regression
- How can this scheme be attacked?
Speeding-up signature verification

- Aggregate signatures
  - Schnorr (CoSi)
  - BLS
- Batch verification
- Screening
Conclusion

• Privacy preserving data collection
  • Accurate results
  • Millions of endpoints
  • Communication through malicious central party
• Our scheme has appealing properties for practical settings
• Scalable tested implementation
Q?