Modern privacy-friendly computing

Dr George Danezis
(g.danezis@ucl.ac.uk)
The “easy” privacy problem: Hiding information from third parties

- Alice and Bob trust each other with their “private” information.
- They wish to hide their interactions from third parties:
  - Encryption hides content.
  - Anonymous communications hide meta-data.
- A relatively well-understood problem.
  - Widely deployed (TLS, Tor).
The “hard” privacy problem: Hiding information from your partners

- Example: “The Millionaire's problem” (Yao)
- Alice and Bob do not trust each other with their secrets, but still want to jointly compute on them.
- Associated problem: they may not trust each other to perform any computations correctly.
The formal problem

- Consider a function \( f \) with \( n \) inputs \( x_i \) from distinct parties returning a result: \( y = f(x_1, \ldots, x_n) \)
  - Correctness: We want to compute \( y \)
  - Privacy: do not learn anything more about \( x_i \) than what we would learn by learning \( y \). Despite the observations \( o \) from the protocol

- In terms of probability:
  - \( \Pr[x_i \mid o, y, x_j] = \Pr[x_i \mid y, x_j] \)
Straw-man Solution: Trusted Third Party

TTP: Every participant has to trust TTP for confidentiality, integrity and availability.
What is wrong with Trusted Third Parties

- May not exist!

- Even if it may exist: The 4 Cs
  - **Cost**: what is the business model? How to implement cheaply?
  - **Corruption**: How do you really know that it will not side with the adversary?
  - **Compulsion**: Legal or extra-legal compulsion to reveal secrets.
  - **Compromise**: It may get hacked!

- Conclusion:
  - TTP: not a robust implementation strategy.
  - However: a great specification strategy (ideal functionality).
No Trusted third party →
Private Computations are impossible!?

• Aim of this talk:
  – Convince you that you can actually compute on secret data, without learning the inputs to the computation.

  – How?
    • Example of linear computations.
    • Understanding of what is missing for a complete system.
Theory:
“Any function can be computed privately without a TTP”

- Even without a coordinator.
- Participants do not learn other's secrets.
  - Can be made robust to cheating.
  - Robust to stopping / failures.

- Two adversary models:
  - Honest but curious: adversary executes protocols correctly but tries to learn as much as possible. ($\frac{1}{2} N + 1$ honest)
  - Byzantine: will send, or drop arbitrary messages to learn the secrets. ($\frac{2}{3} N + 1$ honest)

- Both can be tolerated, but with different efficiency.
How does one prove this generic result?

- **Computation theory:**
  - NAND is sufficient to represent any boolean circuit.
  - NAND can be expressed using the algebraic expression:
    \[ \text{NAND}(A,B) = 1 - AB \]
  - If we can express binary digits, compute addition and multiplication privately, we can compute all circuits.
Two approaches

- Homomorphic encryption:
  - Express 0,1 as ciphertexts $E(0)$, $E(1)$.
  - Allow for operations on ciphertexts producing the cipher text of an addition and multiplication.
  - Here in depth: additive homomorphism only.

- Secret sharing:
  - Express 0,1 as “shares” distributed between users.
  - Do addition and multiplication using protocols on shares.
  - Here in depth: SPDZ addition and multiplication.
Homomorphic Encryption
Homomorphic encryption
The Big Picture

Alice and Bob encrypt their inputs

E(x₁)

Homomorphic operations on ciphertexts

f

Output ciphertext of results

E(y)  Not “y”

E(x₂)
Additively homomorphic public-key encryption

• Goal – define functions for:
  – GenKey
  – Encrypt
  – Decrypt
  – Add
  – (no multiply)

• Note:
  – Add n times is multiplication with a public constant
Mathematical reminder

- Define two elements \( g, h \) that are generators of a cyclic group within which the discrete logarithm problem is believed to be hard.
  - Generators means: \( g^i \) may lead to all group elements.

**Discrete logarithm problem:**
- Given \( g, x \rightarrow g^x \) is easy to compute.
- Given \( g, g^x \rightarrow x \) is hard to compute.

**Security assumption.**

- Example such groups:
  - Integers modulo a prime ("mod p"). (>1024 bits)
  - Points on Elliptic curves. (>160 bits)
The Benaloh Crypto-system (1)

Key Generation

- First introduced in the context of e-voting, to count votes.
- The Scheme:
  - Public: $g$, $h$
  - Key generation: generate a random “$x$”; Private key is “$x$”, Public key is $pk = g^x$. 
The Benaloh Crypto-system (2)

Encryption

• Encryption: 2 elements
  – Encryption of \( m \) with \( pk \):
    random \( k; \)
    \[
    E(m; k) = (g^k, pk^k h^m)
    \]
    If \( k \) not known this is like random, totally hiding \( h^m \)
The Benaloh Crypto-system (3)  
Decryption

- Decryption:
  - Decryption of \((a,b)\) with \(x\):
    \[
    h^m = b \left(a^x\right)^{-1} = g^{xk} h^m / g^{xk} = h^m
    \]
    \[
    m = \log_h(h^m)
    \]

- But is \(\log_h\) not hard to compute?
  - Make a table for all small (16-32 bit) values.
The additive homomorphism - Addition

Reminder:
\[ E(m; k) = (g^k, pk^k h^m) \]

- **Homomorphism**
  - Addition of
    \[ E(m_0; k_0) = (a_0, b_0) \text{ and } E(m_1; k_1) = (a_1, b_1) \]
    \[ E(m_0 + m_1; k_0 + k_1) = (a_0 a_1, b_0 b_1) \]

- Why does that work?
  - \( (a_0 a_1, b_0 b_1) = (g^{k_0} g^{k_1}, g^{xk_0} h^{m_0} g^{xk_1} h^{m_1}) = (g^{k_0+k_1}, g^{x(k_0+k_1)} h^{m_0+m_1}) \)
The additive homomorphism - Multiplication by a **public** constant

- Homomorphism
  - Multiplication of
    \[ E(m_0; k_0) = (a_0, b_0) \text{ with a} \]
    public constant \( c \):
    \[ E(cm_0; ck_0) = ((a_0)^c, (b_0)^c) \]

- Why it works:
  - \((a_0)^c, (b_0)^c\) = \((g^{ck_0}, \text{pub}^{ck_0} h^{cm}) = E(cm_0; ck_0)\)

- Not sufficient for all operations.
  (No multiplication of secrets)
Application 1: Simple Statistics

• **Problem:** A poll asks a number of participants whether they prefer “red” or “blue”. How many said “red” and how many “blue”?

• **Solution:** Each participant submits a Benaloh ciphertext for both “red” and “blue” to an authority. The authority can homomorphically add them.
## Illustrated

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
<th>...</th>
<th>Zoe</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>E(0)</td>
<td>E(1)</td>
<td>...</td>
<td>E(1)</td>
<td>E(10)</td>
</tr>
<tr>
<td>E(1)</td>
<td>E(0)</td>
<td>...</td>
<td>E(0)</td>
<td>E(5)</td>
</tr>
</tbody>
</table>

**Authority**

**Compute ...**
Discussion

• Domain of plaintext is small (up to number of participants), so decryption by enumeration is cheap.

• The Key question: Who's public key? Who has the decryption key?

• The Decryption question: Who decrypts?
  – If single entity → TTP!
  – If no-one: scheme is useless! (Outsourced computation?)
Threshold Decryption

• Answer: it is better if no one has the secret key.
  – No TTP!

• Threshold decryption:
  – The secret key is distributed across many different people.
  – Each have to contribute to the decryption.
  – Even if one is missing, remaining cannot decrypt.
Threshold Decryption – how?

• How?
  – Private keys: $x_1, \ldots, x_n$
  – Public key: $g^{x_1+\ldots+x_n}$
  – Decryption of (a,b): $hm = b / a^{x_1} / a^{x_2} / \ldots / a^{x_n}$

• Why this works?
  – $(((b / a^{x_1}) / a^{x_2}) / \ldots / a^{x_n}) = b / a^{x_1+\ldots+x_n} = hm$
Beyond the Benaloh limitations

- Raw RSA:
  - Multiplicative homomorphism
  - No addition :-(

- Paillier Encryption:
  - Additive homomorphism only
  - Based on RSA: large ciphertexts, slow

- Schemes based on Pairings on Elliptic curves:
  - Addition and 1 multiplication!

  ...

- Breakthrough: Gentry (2009) A fully homomorphic scheme
  - Extremely inefficient! But cool.

- Somewhat Homomorphic Schemes (SHE):
  - Vinod Vaikuntanathan et al.
  - Larger ciphertexts (30Kb), but fast operations (Add 1ms, Mult 50ms)
  - Variable but limited circuit depth.
What is cool about homomorphic schemes?

• Simple architecture:
  – Everyone just provides encrypted inputs. One party (any) computes the function.

• Secret functions:
  – Parts of the function itself may remain secret. The service can perform whatever operations without telling any party.

• Powerful and efficient:
  – Any function of shallow depth.
  – Linear operations are very fast. (Order one field multiplications)
  – Multiplications can be fast-ish (for SHE)
The downsides of homomorphisms

• Expressiveness:
  – Expressing computations as boolean circuits makes them much more expensive (example: no binary search!)

• Efficiency:
  – Every bit → 160bit, 1024bits, …, 30Kbs.

• The problem of decryption (Part 2): Integrity
**Attack:** What is the party doing the computation is actively malicious?

Alice and Bob encrypt their inputs

\[ E(x_1) \quad E(x_2) \]

Homomorphic operations on ciphertexts

\[ f(x_1, x_2) \]

Output ciphertext of results

\[ E(y) \]

Threshold Decryption

\[ y \]

\[ x_1 \]

**Attack:** A malicious party can simply ask the threshold decryption parties to decrypt a secret, not the output of the computation! (Trade name: a decryption oracle attack)

**Lesson:** No confidentiality without integrity!
Attack: Integrity and cheating?

Alice and Bob encrypt their inputs

\[ E(z) \]
\[ E(x_1) \]
\[ E(x_2) \]

Homomorphic operations on ciphertexts

Output ciphertext of results

Threshold Decryption

\[ E(y') \]

\[ y' \]
\[ y \]
No confidentiality without integrity!

• What to do?
  – The central party needs to prove that the output of the computation was indeed correct.
  – Easy case: computation is public, anyone can verify it
    • Ouch. Expensive.
    • Techniques to verify correctness of outsourced computations.
  – Hard case: computation is private.
    • No one has really dealt with this case.
    • Maybe: if private information can be turned into data? …
Secret sharing
Secret Sharing based private computations

• The core idea:
  – Each secret is “shared” across many authorities.
  – Those authorities use protocols to transform shares of secrets into shares of function of secrets.
  – Key: addition & multiplication

• SPDZ variant:
  – Pre-computations to speed up multiplication (using SHE)
  – Integrity protection, nearly for free!
Architecture

\[ f(x_1, x_2) \]
Secret Sharing: pros and cons

• Pros:
  – Well understood complete protocols.
  – Integrity can be very cheap.
  – Actual operations are very cheap.

• Cons:
  – Network interactions.
  – Vast number of pre-computations (triplets – one per gate).
  – Circuits express inefficiently.
  – Computations cannot be secret!
Overall conclusions

- Private computations:
  - You can do any computation privately.
  - It will cost you.
- Linear operations are cheap.
- Non-linear operations less so.
- Limited non-linear depth helps a lot with efficiency.

- Integrity:
  - A problem for confidentiality.

- Maturity:
  - Tool chains and compilers: research grade.
  - Too slow to use for bulk computations.
  - Special high-value computations OK – i.e. billing.
  - Use it to implement functions of the TCB securely.
Appendix on Secret Sharing
The basic scheme

- We work in the field of integers modulo a prime $p$
  - Clock arithmetic with “$p$ hour” clock.

- A share of secret “$x$” is denoted “$<x>$”
  - If we add all shares “$<x>$” (mod $p$) we get “$x$”

- Toy example:
  - Prime $p = 2$, $x = 1$
  - Shares $<x>$ are $\{1, 1, 0, 1, 0\}$
  - Check: $1 + 1 + 0 + 1 + 0 \mod 2 = 1$
Addition of secrets is simple!

• Sharing is based on addition:
  – Natural additive homomorphism.

• Add \(<a>\) and \(<b>\):
  – Each authority can simply add the shares
  – \(<c> = <a+b> = <a> + <b> \mod p\)
  – No distributed protocol is necessary.
Public constant addition and multiplication

- **Add** \(<a>\) to a constant \(k\):
  - Split \(k\) into \(<k>\) as \(\{0,0,...,0,k\}\)
  - Do addition between \(<k>\) and \(<a>\)

- **Multiply** \(<a>\) by a public constant \(k\):
  - Each authority privately computes (no interaction)
  - \(<c> = <ka> = k<a>\)
Multiplication of secrets

• More complex:
  – Need some pre-computed values.
  – Interactive protocol between authorities.

• Pre-computed values:
  – Independent from the function “f”.
  – Can be batch produced beforehand.
Multiplication

- Precomputed triples: \( <a>, <b>, <c> \)
  - Such that \( <c> = <ab> \)

- Protocol to multiply \( <x> \) and \( <y> \):
  - Get fresh pre-computed triplet \( <a>,<b>,<c> \)
  - Compute
    \[
    <e> = <x> + <a> \\
    <d> = <y> + <b>
    \]
  - Publish \( <e> \) and \( <d> \) to get \( e \) and \( d \).
  - Compute:
    \[
    <z> = <xy> = <c> - e<b> - d<a> + ed
    \]

Note: \( a, b \) are randomly distributed so they totally hide \( x \) and \( y \)

Linear!
Logic gates

• Share secret input bits <0> or <1>
• Define function f as a circuit
• Boolean gates:
  – NOT(a) = 1 – a
  – AND(a, b) = ab
    • NAND(a, b) = 1 – ab
  – NOR(a, b) = (1 – a) (1 – b)
  – XOR(a, b) = (a-b)^2
The problem with circuits

- Doing an addition of a 32 bit number:
  - Multiplicative depth of about 14.
  - Requires many rounds of interaction.

- It is much faster to do linear operations on shares of the actual secrets rather than bits.

- Solution:
  - Protocol to convert shares of bits to full representations.
    eg. \langle 1 \rangle, \langle 1 \rangle to \langle 3 \rangle
  - Protocol to convert a secret share to its bit representation
    eg. \langle 3 \rangle to \langle 1 \rangle, \langle 1 \rangle
What about integrity?

• Why do we need integrity?
  – Authorities could be malicious
  – Threat: they wish to change the result.
  – Threat: they wish to leak information about the secrets by not following the protocol.

• Traditional approach:
  – Each authority performs a zero-knowledge proof that what it publishes is correct.
  – Downside: expensive process.
SPDZ integrity

- Use a Message Authentication Code
  - Associate the share of a MAC with each secret share.
  - Maintain the MAC through computations.
  - Never reveal the MAC!
  - However, check that it is correct.

- SPDZ MACs:
  - MAC key is a secret $v$ shared as $<v>$
  - Each share $<a>$ has a MAC share $<va>$
  - Protocol to take $<v>$ and endorse it to provide $<v>$

- Authorities know $<v>$ but no one ever knows $v$
Operations with MACs

- Addition just works by adding secrets and MACs.

- Multiplication:
  - Pre-shared values need to have a MAC.
  - Otherwise it is the same technique, for secret and MAC.

- Constants:
  - Easy to endorse them.
  - For k compute $<k>, <vk> = k<v>$
How to check the MAC without revealing it?

• Intuition:
  – Every value has a MAC associated with it.
  – Operations preserve the MAC.
  – Everything that is declassified needs to have its MAC checked.
  – This is the point where authorities interact!
    • Threat model: one authority can give the wrong share.
  – Check that the relation holds:
    • For fixed MAC key $<v>$ check that $a, <va>$
    • Relation $a<v> = <va>$
  – Without revealing $<v>$. 
How?

• Note that:
  – $k<v> == <kv>$
  – Same as $(k<v> - <kv> = <0>)$
  – Same as $w (k<v> - <kv>) = <0>$
    • For a randomly distributed “$w$”

• Can do many in parallel!
  – $\sum_i w_i (k_i<v> - <k_i<v>) = <0>$
Integrity protocol (outline)

- Perform all operations
- Commit to intermediate results and final results (Do not reveal final results)
- Jointly generate random $w_i$
- For all intermediate results compute:
  - $<c> = \sum_i w_i (k_i <v> - <k_i<v>) = <0>$
  - Reveal $<c>$ and check it is zero!
  - That guarantees no information leaks from the secrets, since computations are correct until the end.
- Reveal results:
  - $<c> = \sum_i w_i (k_i <v> - <k_i<v>) = <0>$
  - Reveal $<c>$ and check it is zero!
  - That guarantees the actual results are also correct.
The cost of integrity

- Low.
- Two shares instead of one.
- Triplets with MACs for multiplications.
- Two checks per computation
  - That are batched.