

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



Who is richer?

I am also curious but I do not want to tell you how much I earn.

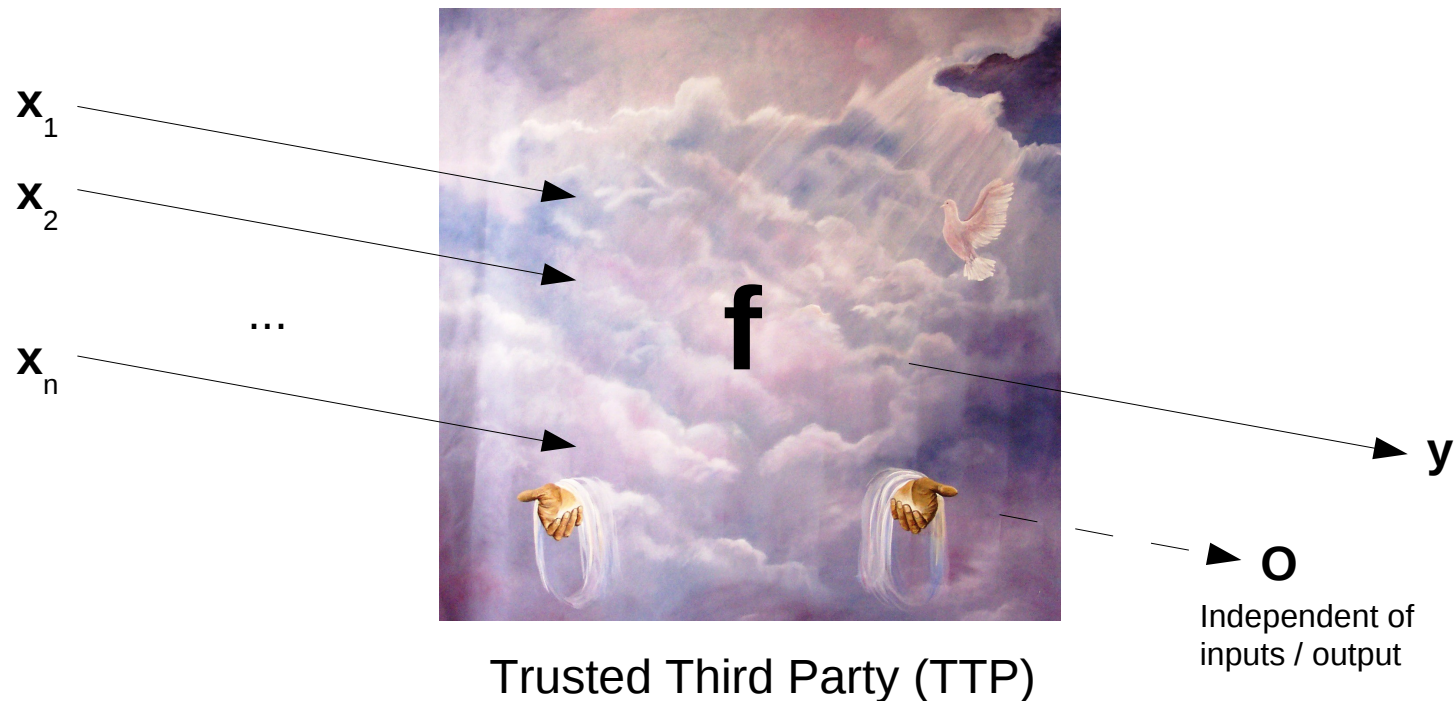


- Example: “The Millionaire's problem” (Yao)
- Alice and Bob do not trust each other with their secrets, but still want to jointly compute on them.
- Associated problem: they may not trust each other to perform any computations correctly.

The formal problem

- Consider a function \mathbf{f} with \mathbf{n} inputs \mathbf{x}_i from distinct parties returning a result: $\mathbf{y} = \mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n)$
 - Correctness: We want to compute \mathbf{y}
 - Privacy: do not learn anything more about \mathbf{x}_i than what we would learn by learning \mathbf{y} . Despite the observations \mathbf{o} from the protocol
- In terms of probability:
 - $\Pr[\mathbf{x}_i \mid \mathbf{o}, \mathbf{y}, \mathbf{x}_j] = \Pr[\mathbf{x}_i \mid \mathbf{y}, \mathbf{x}_j]$

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.

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

- 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.

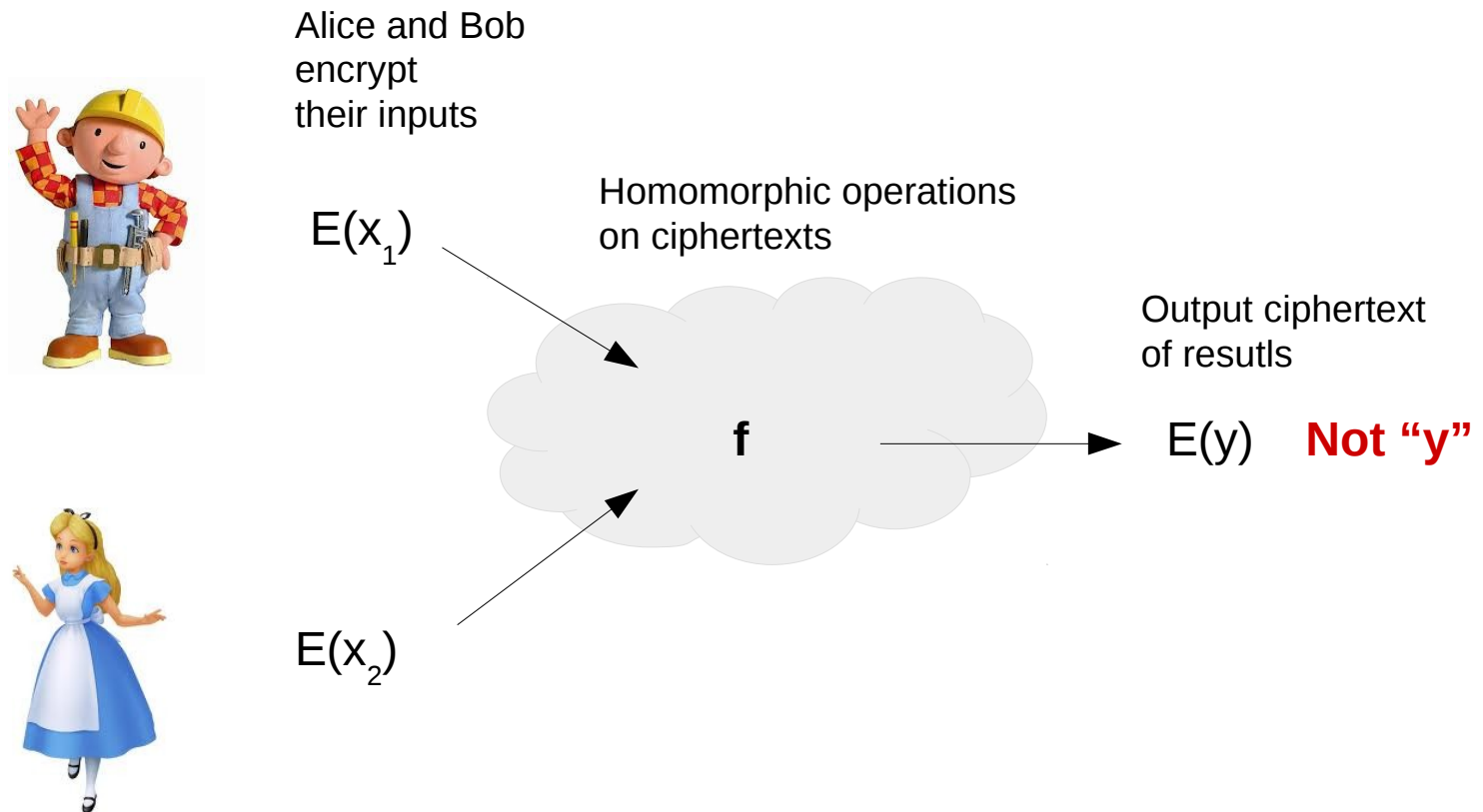
Two approaches

- 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



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 :

$$hm = b (a^x)^{-1} = g^{xk} h^m / g^{xk} = h^m$$

$$m = \log_h(hm)$$

- 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?

- $(a_0 a_1, b_0 b_1) = (g^{k_0} g^{k_1}, g^{x k_0} h^{m_0} g^{x k_1} h^{m_1}) = (g^{k_0 + k_1}, g^{x(k_0 + k_1)} h^{(m_0 + m_1)})$

The additive homomorphism - Multiplication by a public constant

- Homomorphism

- Multiplication of

- $E(m_0; k_0) = (a_0, b_0)$ with a

- public constant c :

- $E(\mathbf{c}m_0; \mathbf{c}k_0) = ((\mathbf{a}_0)^c, (\mathbf{b}_0)^c)$

- Why it works:

- $((\mathbf{a}_0)^c, (\mathbf{b}_0)^c) = (g^{\mathbf{c}k_0}, \text{pub}^{\mathbf{c}k_0} h^{\mathbf{c}m}) = E(\mathbf{c}m_0; \mathbf{c}k_0)$

- 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



...



...



Compute ...

Alice	Bob	...	Zoe	Total
E(0)	E(1)	...	E(1)	E(10)
E(1)	E(0)	...	E(0)	E(5)

Authority

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, \dots, x_n
- Public key: $g^{x_1+\dots+x_n}$
- Decryption of (a,b): $hm = b / a^{x_1} / a^{x_2} / \dots / a^{x_n}$

- Why this works?

- $((b / a^{x_1}) / a^{x_2}) / \dots / a^{x_n} = b / a^{x_1+\dots+x_n} = hm$

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.

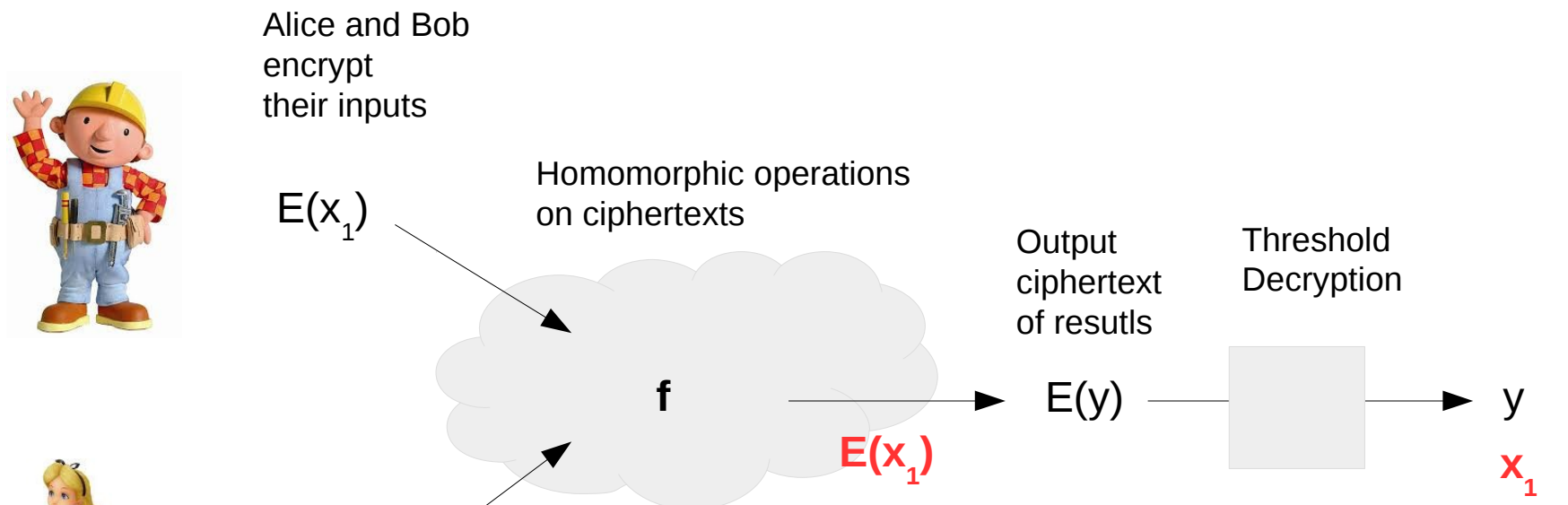
What is cool about homomorphic schemes?

- 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 \rightarrow 160bit, 1024bits, ..., 30Kbs.
- The problem of decryption (Part 2): Integrity

Attack: What is the party doing the computation is actively malicious?



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

$E(z)$

$E(x_1)$

Homomorphic operations
on ciphertexts

f

Output
ciphertext
of results

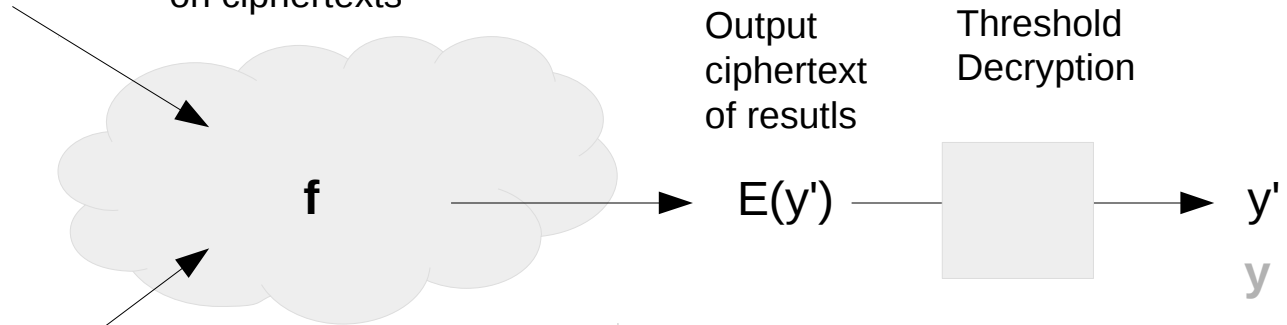
$E(y')$

Threshold
Decryption

y'
 y



$E(x_2)$



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



x_1



x_2

Authority 1

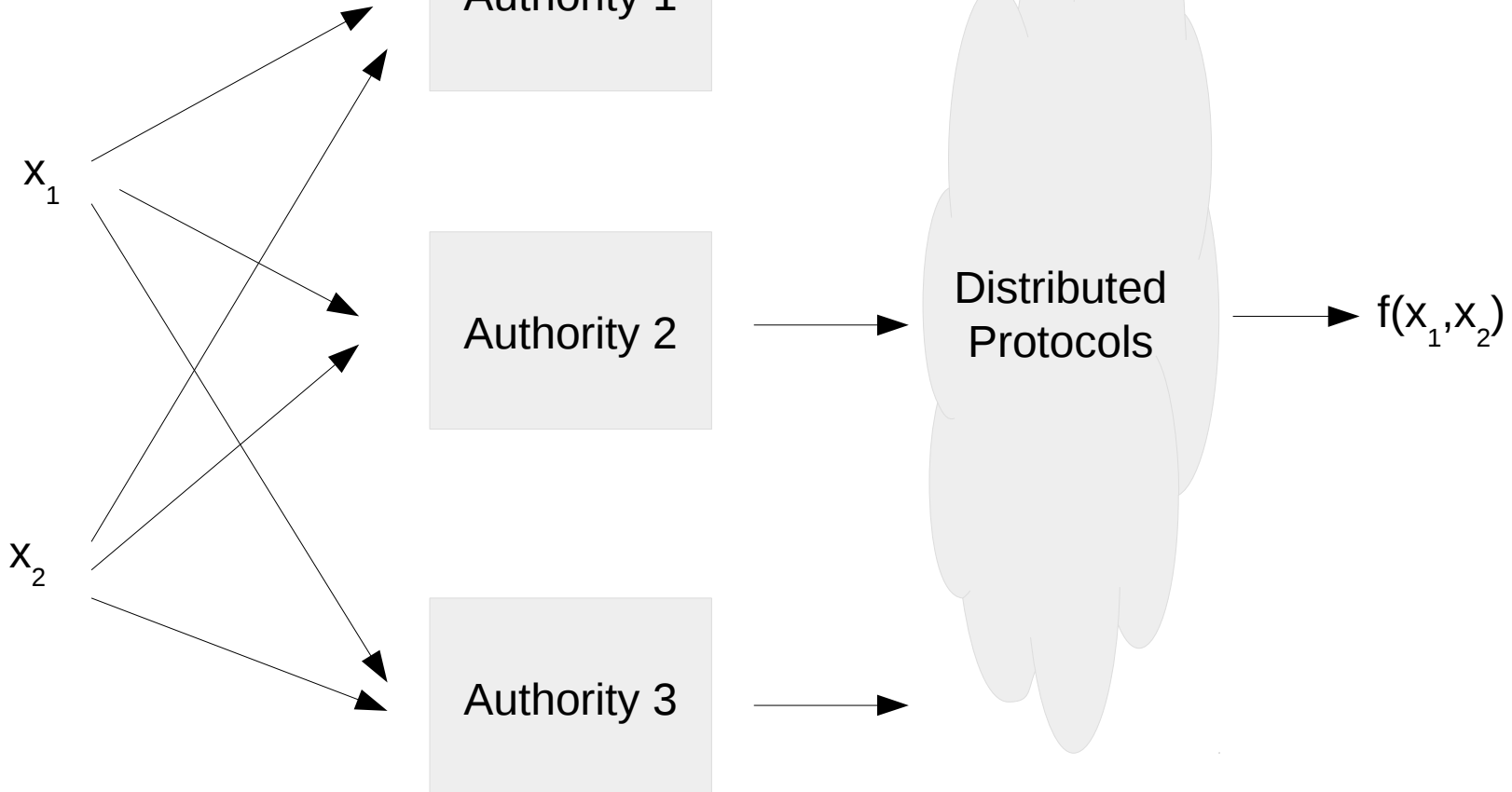
Authority 2

Authority 3



Query (f)

$f(x_1, x_2)$



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.**
 - Compute: homomorphic encryption.
 - Network: secret sharing.
 - 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 “ $\langle x \rangle$ ”
 - If we add all shares “ $\langle x \rangle$ ” (mod p) we get “ x ”
- Toy example:
 - Prime $p = 2$, $x = 1$
 - Shares $\langle x \rangle$ are $\{1, 1, 0, 1, 0\}$
 - Check: $1 + 1 + 0 + 1 + 0 \bmod 2 = 1$

Addition of secrets is simple!

- Sharing is based on addition:
 - Natural additive homomorphism.
- Add $\langle a \rangle$ and $\langle b \rangle$:
 - Each authority can simply add the shares
 - $\langle c \rangle = \langle a+b \rangle = \langle a \rangle + \langle b \rangle \pmod p$
 - No distributed protocol is necessary.

Public constant addition and multiplication

- Add $\langle a \rangle$ to a constant k :
 - Split k into $\langle k \rangle$ as $\{0, 0, \dots, 0, k\}$
 - Do addition between $\langle k \rangle$ and $\langle a \rangle$
- Multiply $\langle a \rangle$ by a public constant k :
 - Each authority privately computes (no interaction)
 - $\langle c \rangle = \langle ka \rangle = k\langle a \rangle$

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.
 - How? Using TTP, Secure Hardware, SHE (SPDZ).

Multiplication

- Precomputed triples: $\langle a \rangle$, $\langle b \rangle$, $\langle c \rangle$
 - Such that $\langle c \rangle = \langle ab \rangle$

- Protocol to multiply $\langle x \rangle$ and $\langle y \rangle$:

- Get fresh pre-computed triplet $\langle a \rangle, \langle b \rangle, \langle c \rangle$

- Compute

$$\langle e \rangle = \langle x \rangle + \langle a \rangle$$

$$\langle d \rangle = \langle y \rangle + \langle b \rangle$$

- Publish $\langle e \rangle$ and $\langle d \rangle$ to get e and d .

- Compute:

$$\langle z \rangle = \langle xy \rangle = \langle c \rangle - e\langle b \rangle - d\langle a \rangle + ed$$

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

Linear!

Logic gates

- Share secret input bits $\langle 0 \rangle$ or $\langle 1 \rangle$
- Define function f as a circuit
- Boolean gates:
 - $\text{NOT}(a) = 1 - a$
 - $\text{AND}(a, b) = ab$
 - $\text{NAND}(a, b) = 1 - ab$
 - $\text{NOR}(a, b) = (1 - a) (1 - b)$
 - $\text{XOR}(a, b) = (a-b)^2$

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 $\langle v \rangle$
 - _ Each share $\langle a \rangle$ has a MAC share $\langle va \rangle$
 - _ Protocol to take $\langle v \rangle$ and endorse it to provide $\langle v \rangle$
- Authorities know $\langle v \rangle$ 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 $\langle k \rangle$, $\langle vk \rangle = k \langle v \rangle$

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 $\langle v \rangle$ check that $a, \langle va \rangle$
 - Relation $a \langle v \rangle = \langle va \rangle$
 - Without revealing $\langle v \rangle$.

How?

- Note that:
 - $k\langle v \rangle == \langle kv \rangle$
 - Same as $(k\langle v \rangle - \langle kv \rangle = \langle 0 \rangle)$
 - Same as $w (k\langle v \rangle - \langle kv \rangle) = \langle 0 \rangle$
 - For a randomly distributed “w”

- Can do many in parallel!
 - $\text{Sum}_i w_i (k_i \langle v \rangle - \langle k_i v \rangle) = \langle 0 \rangle$

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:
 - _ $\langle c \rangle = \text{Sum}_i w_i (k_i \langle v \rangle - \langle k_i v \rangle) = \langle 0 \rangle$
 - _ Reveal $\langle c \rangle$ and check it is zero!
 - _ That guarantees no information leaks from the secrets, since computations are correct until the end.
- Reveal results:
 - _ $\langle c \rangle = \text{Sum}_i w_i (k_i \langle v \rangle - \langle k_i v \rangle) = \langle 0 \rangle$
 - _ Reveal $\langle c \rangle$ 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.