CS 798
Privacy in Computation and Communication

Module 3
Privacy in Computation: Distributed Trust

Spring 2024

Distributed trust

Recall the three main ways to achieve privacy in computation:

- Distributed trust
- Trusted hardware
- Homomorphic encryption

Distributed trust

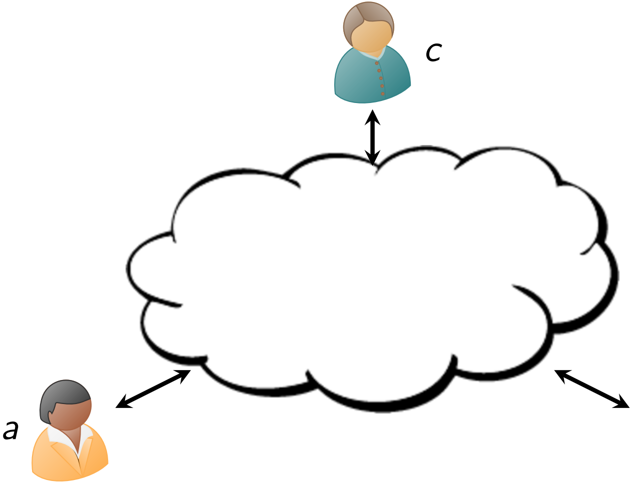
Recall the three main ways to achieve privacy in computation:

- Distributed trust
- Trusted hardware
- Homomorphic encryption

MPC: multiparty computation

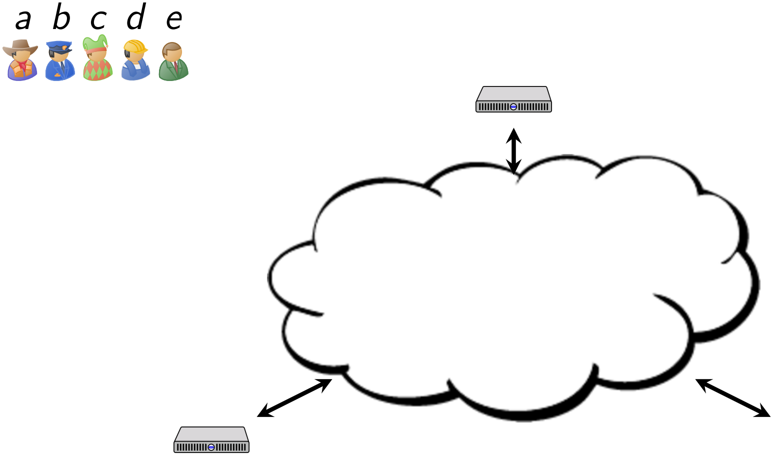
- The main way to use distributed trust to achieve privacy in computation is by using MPC (multiparty computation)
- Sometimes called *SMC* (secure multiparty computation)

MPC: multiparty computation



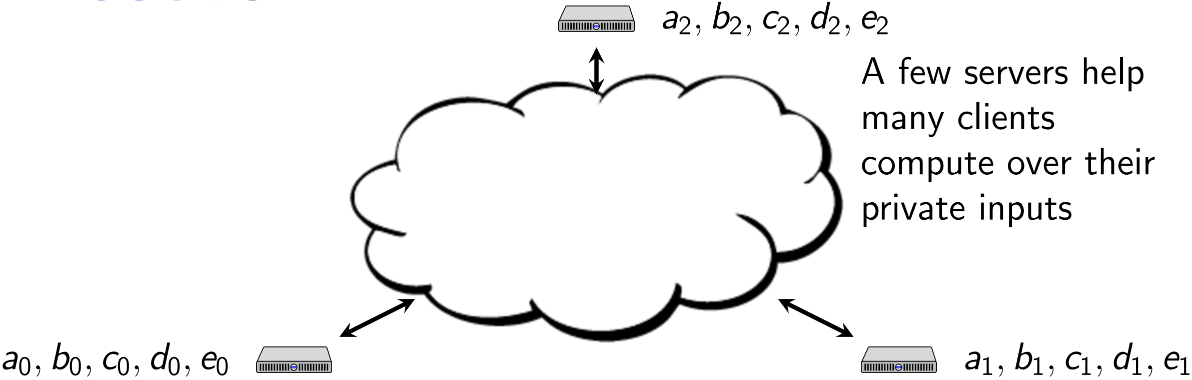
Alice, Bob, and Carol want to compute some function $f(a, b, c)$

MPC: multiparty computation



A few servers help many clients compute over their private inputs

MPC: multiparty computation



A few servers help many clients compute over their private inputs

Properties of MPC protocols

- Expressibility
- Minimum number of parties
- Threat model
- Maximum number of adversarial parties
- Performance

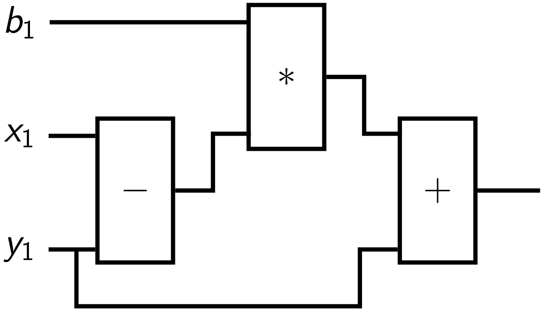
Expressibility

- What functions f can the MPC protocol compute privately?
- Some protocols are *generic*: they can compute *any* function that has bounded runtime
- Some are *specific*: they are designed to (more efficiently) compute one particular function
- In this module, we will start with generic protocols, and later look at a few specific ones

Generic protocols

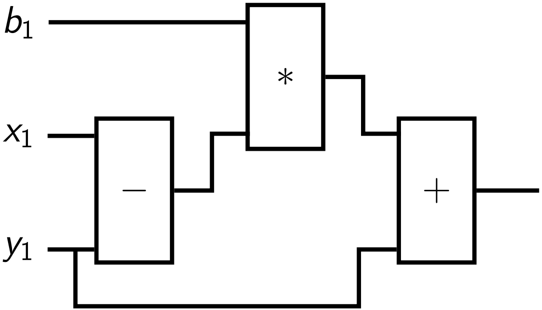
- As discussed in Module 1, the high-level approach is to express your function as a *circuit* of Boolean or arithmetic gates
- Some protocols come with a *compiler* that will take your function written in some reasonable language, and automatically generate the circuit for you
- Recall that circuits are *oblivious*: they always perform the same actions, regardless of the input, since the parties executing the circuit *cannot know the input*
  - So the compiler must compile any if/then/else statements into circuits that compute *both* the "then" and "else" parts, and use the "if" test to select which results to keep and which to discard

Generic protocols



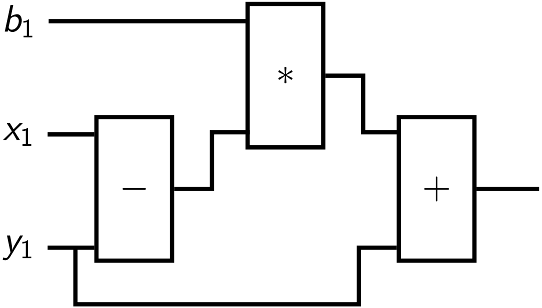
- The clients (with the inputs) secret share their inputs across all the (computational) parties (party 1 shown above)

Generic protocols



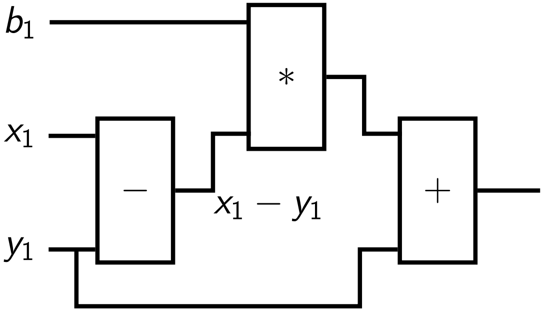
- For each gate in turn, the parties jointly evaluate the gate
- Each party has a share of the inputs to the gate; each party must compute a share of the output of the gate *without learning the true inputs or output*

Generic protocols



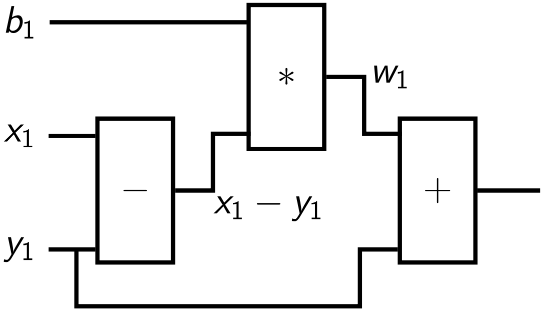
- All of the secret sharing schemes (additive, XOR, Shamir, replicated) we talked about are *linear*
- That means if $x_1, \ldots, x_n$ are shares of a value x and $y_1, \ldots, y_n$ are shares of a value y, then $(x_1 + y_1), \ldots, (x_n + y_n)$ are shares of $x + y$

Generic protocols



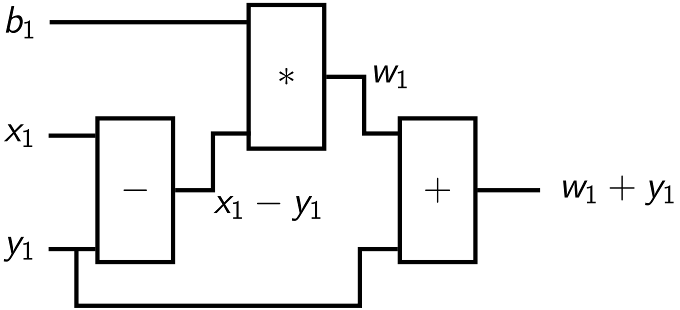
- Linear gates ($\oplus$, $+$, $-$, multiplication by a public value) are then easy:
- Each party just locally computes the gate on its shares of the inputs to get its share of the output

Generic protocols



- Non-linear gates ($\wedge$, $\vee$, $*$) are more complicated, and require the parties to interact for each such gate
- The details vary with each protocol; we'll look at some examples later on

Generic protocols



- Non-linear gates ($\wedge$, $\vee$, $*$) are more complicated, and require the parties to interact for each such gate
- The details vary with each protocol; we'll look at some examples later on

Minimum number of parties

- How many parties do you need to run this protocol?
  - Count the parties participating in the computation, not clients that just submit input values
  - (Enough of) these parties must not collude to reconstruct the private inputs!
  - Distributed trust: you have to trust that *some* of the parties are behaving properly, but you don't need to know *which ones*

- 2, 3, 4 are common values
  - Often written 2-PC, 3-PC, 4-PC

- The larger this value, the more challenging it will be to deploy the protocol
  - You need to find that many parties who will *collaborate* to execute the protocol, but not *collude* to break it

Threat model

- As discussed earlier, some of the parties may be untrustworthy / adversarial
- Does the protocol remain secure if *some* of the parties collude, but otherwise follow the protocol?
  - It typically *cannot* remain secure if *all* parties collude, since then there's effectively just one party, and there's no distributed trust
- Does it remain secure if some of the parties deviate from the protocol?
  - Both: does the adversary learn the private inputs, but also can the adversary crash the protocol and cause it not to output the correct answer (or not output anything at all)?
  - Producing the correct answer even when some parties misbehave is called *robustness*

Maximum number of adversarial parties

- *How many* parties in the protocol can be adversarial and still have the protocol be secure?
- Weakest form: just one; if even two parties collude, they can learn the private inputs
- Strongest form: all but one; if even one party is honest, the private inputs are safe
    - But: it's not generally possible to make such systems robust, so there's a tradeoff

Maximum number of adversarial parties

- There are two broad classes of protocols:
- Honest majority:
  - The number of adversarial parties is strictly smaller than the number of honest parties
  - Example: 3-PC where one party can be adversarial, or 5-PC where two parties can be adversarial
- Honest minority:
  - As few as one party needs to be honest
  - But as above, you generally lose robustness in that case

Performance

- Different MPC protocols have different performance characteristics
- Important things to measure:
  - Local computation at each party
  - Total amount of communication by each party
  - Number of *latencies* / *sequential messages* of communication
- Which is most important?
  - Depends on the deployment scenario

MPC deployment scenarios

- Recall the MPC parties cannot collude
- Imagine all the parties had their machines in a single cloud datacentre (e.g., Amazon)
- Then you're trusting *Amazon itself* not to "peek inside" the running machines to see the shares of the clients' inputs
- If you're willing to do that, why not just have Amazon run one single machine to do the computation without any privacy, and just trust Amazon that it won't look inside?

MPC deployment scenarios

- So for MPC, you need to have machines actually controlled by the different parties
- You *could* have different parties bring their computers all to one place and hook them up together
  - Where no one else has access to the machines
  - This is of course inconvenient and probably unlikely
- But if you can, then you get very fast inter-party communication (tens to hundreds of Gbps) and very low inter-party latencies (tens to hundreds of microseconds)
  - In that case, the bandwidth and number of latencies don't matter very much, and the amount of local computation will dominate

MPC deployment scenarios

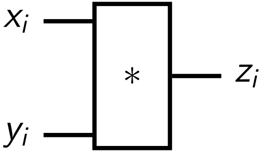
- The alternative is that the parties' machines are communicating over the Internet
- Probably at best ≈1 Gbps, tens of *milliseconds* latency
  - The number of latencies becomes the bottleneck
  - You can do a *lot* of computation in the time it takes to receive a message from another party
- Also note that it's way easier to deploy machines with more computing power (cores, etc.) than it is to increase your bandwidth or decrease your latency to the other parties

Non-linear gates

- We saw earlier that *linear* gates are very easy to evaluate
  - Only some (very simple) local computation, no communication at all
- How do non-linear gates work?
  - It depends on the details of the MPC protocol, and in particular which secret sharing technique is used
- We'll look next at how to compute a multiplication gate, using three different kinds of secret sharing
  - Additive, Shamir, replicated

Multiplication gate

- The general setup is that each party i has shares x_i and y_i of the *inputs* (x and y) to the multiplication gate, and they want to perform some protocol so that each party i ends up with a share z_i of the product $z = x \cdot y$.



Additive secret sharing

- Suppose we have two parties (2-PC) using additive secret sharing
  - So $x = x_1 + x_2$ and $y = y_1 + y_2$
- We want party 1 to end up with z_1 and party 2 to end up with z_2 such that $z_1 + z_2 = x \cdot y = (x_1 + x_2) \cdot (y_1 + y_2)$
  - *Without* revealing x, y, or z to either party!
- The key trick: *Beaver triples*

Beaver triples

- Ahead of time, distribute shares of *random* inputs (a and b) and output (c) of a multiplication gate to the parties
  - So party 1 gets (a_1, b_1, c_1) and party 2 gets (a_2, b_2, c_2), where a_1, b_1, c_1, a_2, b_2 are independent and random, and $c_2 = (a_1 + a_2)\cdot(b_1 + b_2) - c_1$
  - c_2 is also then random (as we saw before), but not independent
- These random triples do not depend on the clients' inputs
- You will need to distribute one Beaver triple in advance for every multiplication gate in the circuit you will want to compute on the clients' inputs

Beaver triples

- The two parties use a and b to *blind* x and y respectively
  - Each party sends their share of $\alpha = x + a$ and $\beta = y + b$ to the other party (so both parties can reconstruct α and β)
  - Since a and b are random, learning $\alpha = x + a$ tells you nothing about x, and similarly for y

- Party 1 computes $z_1 = \alpha y_1 - \beta a_1 + c_1$,
  Party 2 computes $z_2 = \alpha y_2 - \beta a_2 + c_2$

$$
\begin{aligned}
z_1 + z_2 &= \alpha(y_1 + y_2) - \beta(a_1 + a_2) + (c_1 + c_2) \\
&= \alpha \cdot y - \beta \cdot a + c \\
&= (x + a)y - (y + b)a + c \\
&= xy + ay - ay - ab + c = xy \text{ (since } c = ab)
\end{aligned}
$$

Preprocessing

- This protocol is an example of a protocol with a preprocessing phase
- Some amount of work is done in advance, before the clients show up with their inputs
- This can reduce the amount of time it takes to process the clients' inputs once they show up (the *latency*)
- The preprocessing phase is sometimes called the *offline* phase, but that's a bad name
  - The parties definitely have to be online during this phase

Preprocessing

- Where do these Beaver triples come from?
- A couple of options:
- The two parties run an MPC protocol to jointly create them
- Have a third party with a limited role:
  - Only active during the preprocessing phase
  - Just sends a bunch of these random triples to the two parties (in a single latency), and then exits (nothing is ever sent *to* this party)
  - This is sometimes called "2+1-PC" meaning it's 2-PC plus this one more party with the very limited role

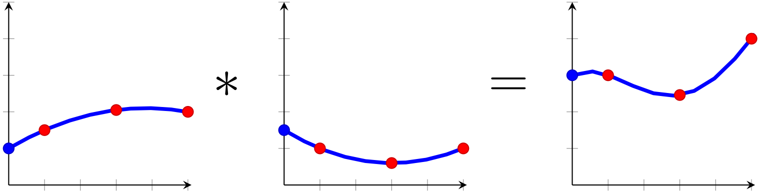
Properties of this protocol

- Expressibility: generic
- Minimum number of parties: 2 (+ 1 preprocessing only)
- Threat model: semi-honest
- Maximum number of adversarial parties: 1
- Performance (g total gates, m mult gates, mult depth d):
  - Local computation: $\mathcal{O}(g)$
  - Total communication: $6m$ preproc + $2m$ per party
  - Latencies: 1 preproc + d

Shamir secret sharing

- With Shamir secret sharing, there are n parties, and any t of them can reconstruct the private data
  - So at most $t-1$ can be adversarial

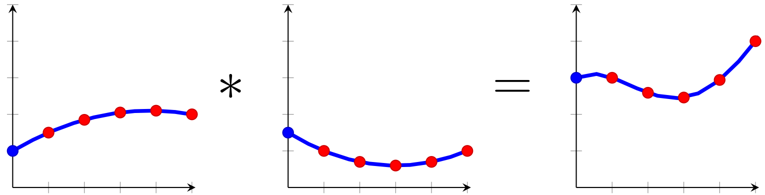
- Recall: shares of a value are points on a degree $t-1$ polynomial whose y-intercept is the value



Shamir secret sharing

- With Shamir secret sharing, there are n parties, and any t of them can reconstruct the private data
  - So at most $t-1$ can be adversarial

- Recall: shares of a value are points on a degree $t-1$ polynomial whose y-intercept is the value



Degree reduction

- If each party i locally multiplies their x_i and y_i to get w_i, then the w_i do lie on a polynomial whose intercept is in fact $x \cdot y$
  - But the degree of that polynomial is $2t - 2$ instead of $t - 1$
- If we *were* to reconstruct the value from the w_i shares, how would we do it?

  $\Rightarrow$ Lagrange interpolation: $w = \lambda_1 w_1 + \lambda_2 w_2 + \cdots + \lambda_n w_n$
- So we want to *privately* compute w from the n private inputs $w_1, \ldots, w_n$ (the λ_i are public, remember)

Degree reduction

- The key trick: we can use MPC for this!
  - And since the Lagrange interpolation formula is linear, we don't have a problem where in order to evaluate a multiplication gate, we need to evaluate one or more multiplication gates
- So the multiplication gate protocol for Shamir secret sharing is:
  - Each party i locally multiplies $x_i \cdot y_i$ to get w_i
  - Each party i makes n shares $w_{i,1}, \ldots, w_{i,n}$ of w_i with the correct t and for each j, sends share $w_{i,j}$ to party j
  - Each party j locally combines the shares they received with Lagrange interpolation to get $z_j = \lambda_1 w_{1,j} + \lambda_2 w_{2,j} + \cdots + \lambda_n w_{n,j}$
  - The z_j are now Shamir secret shares (with the correct t) of $z = x \cdot y$

Degree reduction

- For this to work, we must have enough parties to be able to reconstruct the intercept of the degree $2t - 2$ polynomial
  - So $n \geq 2t - 1$, and recall there are at most $t - 1$ adversarial parties

  $\Rightarrow$ Honest majority setting

- Look what we did here:
  - We evaluated the reconstruction function *using the private computation mechanism itself* in order to get a "clean" sharing of a value
  - We will see this technique again later in the course

Properties of this protocol

- Expressibility: generic
- Minimum number of parties: $n \geq 2t - 1$
- Threat model: semi-honest
- Maximum number of adversarial parties: $t - 1$
- Performance (g total gates, m mult gates, mult depth d):
  - Local computation: $\mathcal{O}(g + ntm)$
  - Total communication: $(n - 1)m$ per party
  - Latencies: d

Replicated secret sharing

- Recall how replicated secret sharing works (simple example: $n = 3$, $t = 2$)
  - Each value is additively shared into 3 pieces, each party gets 2 of them
  - $x = x_1 + x_2 + x_3$, $y = y_1 + y_2 + y_3$
  - Party 1 gets: (x_1, x_2), (y_1, y_2)
  - Party 2 gets: (x_2, x_3), (y_2, y_3)
  - Party 3 gets: (x_3, x_1), (y_3, y_1)

Replicated secret sharing

- Recall how replicated secret sharing works (simple example: $n = 3$, $t = 2$)
  - Each value is additively shared into 3 pieces, each party gets 2 of them
  - $x = x_1 + x_2 + x_3$, $y = y_1 + y_2 + y_3$, want $z_1 + z_2 + z_3 = x \cdot y$
  - Party 1 gets: (x_1, x_2), (y_1, y_2), wants (z_1, z_2)
  - Party 2 gets: (x_2, x_3), (y_2, y_3), wants (z_2, z_3)
  - Party 3 gets: (x_3, x_1), (y_3, y_1), wants (z_3, z_1)

Replicated secret sharing

- First attempt (not quite good enough):
- Want z_1, z_2, z_3 such that

$$
\begin{aligned}
z_1 + z_2 + z_3 = \; & x \cdot y = (x_1 + x_2 + x_3)(y_1 + y_2 + y_3) \\
= \; & x_1 y_1 + x_1 y_2 + x_2 y_1 \\
& + x_2 y_2 + x_2 y_3 + x_3 y_2 \\
& + x_3 y_3 + x_1 y_3 + x_3 y_1
\end{aligned}
$$

Replicated secret sharing

- First attempt (not quite good enough):
- Want z_1, z_2, z_3 such that

$$
\begin{aligned}
z_1 + z_2 + z_3 = \;& x \cdot y = (x_1 + x_2 + x_3)(y_1 + y_2 + y_3) \\
= \;& x_1 y_1 + x_1 y_2 + x_2 y_1 \leftarrow \text{party 1 can compute this} \\
& + x_2 y_2 + x_2 y_3 + x_3 y_2 \\
& + x_3 y_3 + x_1 y_3 + x_3 y_1
\end{aligned}
$$

Replicated secret sharing

- First attempt (not quite good enough):

- Want z_1, z_2, z_3 such that

$$\begin{aligned} z_1 + z_2 + z_3 = \; & x \cdot y = (x_1 + x_2 + x_3)(y_1 + y_2 + y_3) \\ = \; & x_1 y_1 + x_1 y_2 + x_2 y_1 \leftarrow \text{party 1 can compute this} \\ & + x_2 y_2 + x_2 y_3 + x_3 y_2 \leftarrow \text{party 2 can compute this} \\ & + x_3 y_3 + x_1 y_3 + x_3 y_1 \leftarrow \text{party 3 can compute this} \end{aligned}$$

Replicated secret sharing

- First attempt (not quite good enough):
- Want z_1, z_2, z_3 such that

$$\begin{aligned} z_1 + z_2 + z_3 = \; & x \cdot y = (x_1 + x_2 + x_3)(y_1 + y_2 + y_3) \\ = \; & x_1 y_1 + x_1 y_2 + x_2 y_1 \leftarrow z_1 \\ & + x_2 y_2 + x_2 y_3 + x_3 y_2 \leftarrow z_2 \\ & + x_3 y_3 + x_1 y_3 + x_3 y_1 \leftarrow z_3 \end{aligned}$$

Replicated secret sharing

- First attempt (not quite good enough):

- Want z_1, z_2, z_3 such that

$$\begin{aligned} z_1 + z_2 + z_3 = \;& x \cdot y = (x_1 + x_2 + x_3)(y_1 + y_2 + y_3) \\ = \;& x_1 y_1 + x_1 y_2 + x_2 y_1 \leftarrow z_1 \\ & + x_2 y_2 + x_2 y_3 + x_3 y_2 \leftarrow z_2 \\ & + x_3 y_3 + x_1 y_3 + x_3 y_1 \leftarrow z_3 \end{aligned}$$

- Then party 1 sends z_1 to party 3, party 2 sends z_2 to party 1, party 3 sends z_3 to party 2

Replicated secret sharing

- First attempt (not quite good enough):

- Want z_1, z_2, z_3 such that

$$
\begin{aligned}
z_1 + z_2 + z_3 = \;& x \cdot y = (x_1 + x_2 + x_3)(y_1 + y_2 + y_3) \\
= \;& x_1 y_1 + x_1 y_2 + x_2 y_1 \leftarrow z_1 \\
+ \;& x_2 y_2 + x_2 y_3 + x_3 y_2 \leftarrow z_2 \\
+ \;& x_3 y_3 + x_1 y_3 + x_3 y_1 \leftarrow z_3
\end{aligned}
$$

- Problem: Party 3 (for example) is supposed to learn z_1 but already knows x_1 and y_1, and so can learn information about x_2 and y_2

Zero sharing

- The key trick: non-interactive *zero sharing*
  - The parties can, *without communication*, come up with random α_1, α_2, α_3 such that $\alpha_1 + \alpha_2 + \alpha_3 = 0$
  - Use those α_i to *blind* the values on the previous slide to prevent the information leakage:

$$\begin{aligned}
\text{Party 1 computes } z_1 &= x_1 y_1 + x_1 y_2 + x_2 y_1 + \alpha_1 \\
\text{Party 2 computes } z_2 &= x_2 y_2 + x_2 y_3 + x_3 y_2 + \alpha_2 \\
\text{Party 3 computes } z_3 &= x_3 y_3 + x_1 y_3 + x_3 y_1 + \alpha_3
\end{aligned}$$

  - Then party 1 sends z_1 to party 3, party 2 sends z_2 to party 1, party 3 sends z_3 to party 2

Zero sharing

- So how do the parties make these α_i values?

- Remember PRGs: given a key as input, produce an arbitrary-length sequence of random-looking outputs

- Ahead of time, each party i picks a random PRG key k_i
  - Party 1 sends k_1 to party 3, party 2 sends k_2 to party 1, party 3 sends k_3 to party 2

- When the parties want new α_i values, they compute r_i as the next output of $\mathrm{PRG}(k_i)$
  - Party 1 knows (r_1, r_2), computes $\alpha_1 = r_1 - r_2$
  - Party 2 knows (r_2, r_3), computes $\alpha_2 = r_2 - r_3$
  - Party 3 knows (r_3, r_1), computes $\alpha_3 = r_3 - r_1$

Properties of this protocol

- Expressibility: generic
- Minimum number of parties: 3
- Threat model: semi-honest
- Maximum number of adversarial parties: 1
- Performance (g total gates, m mult gates, mult depth d):
  - Local computation: $\mathcal{O}(g)$
  - Total communication: 3 preproc $+$ m per party
  - Latencies: 1 preproc $+$ d

Protocols for specific functions

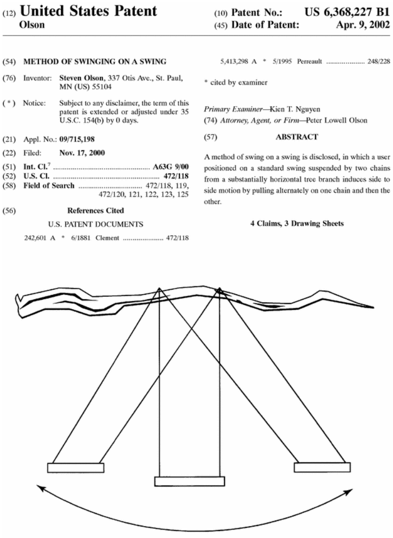
- We next turn our attention to MPC protocols for specific (not generic) functions
- These can often be implemented more efficiently than by implementing the function using generic MPC
- We will look at a few such MPC protocols for specific functions
  - Private information retrieval
  - Private set intersection
  - Threshold signatures

Private information retrieval

- You want to look something up in an online database
  - For example, a database of patents
- You want to keep private *the information being retrieved*
  - For example, the patent number (6368227) you're looking up

Private information retrieval

- You want to look something up in an online database
  - For example, a database of patents
- You want to keep private *the information being retrieved*
  - For example, the patent number (6368227) you're looking up



(12) **United States Patent**
Olson

(10) **Patent No.: US 6,368,227 B1**
(45) **Date of Patent: Apr. 9, 2002**

(54) **METHOD OF SWINGING ON A SWING**

(76) Inventor: **Steven Olson**, 337 Otis Ave., St. Paul, MN (US) 55104

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/715,198**

(22) Filed: **Nov. 17, 2000**

(51) **Int. Cl.**[7] **A63G 9/00**
(52) **U.S. Cl.** **472/118**
(58) **Field of Search** 472/118, 119, 472/120, 121, 122, 123, 125

(56) **References Cited**

U.S. PATENT DOCUMENTS

242,601 A * 6/1881 Clement 472/118
5,413,298 A * 5/1995 Perreault 248/228

* cited by examiner

Primary Examiner—Kien T. Nguyen
(74) *Attorney, Agent, or Firm*—Peter Lowell Olson

(57) **ABSTRACT**

A method of swing on a swing is disclosed, in which a user positioned on a standard swing suspended by two chains from a substantially horizontal tree branch induces side to side motion by pulling alternately on one chain and then the other.

4 Claims, 3 Drawing Sheets

Private information retrieval

- Other uses include:
  - Looking up whether a password is in a list of breached credentials (without revealing the password)
  - Looking up whether a URL is in a list of malicious websites (without revealing the URL)

- This is called *private information retrieval* (PIR)
  - Simplest form: you know the exact record number you want to look up (e.g., patent number)
  - But can also do more advanced queries, such as query by (private) keyword, or even SQL queries (where the prepared statement is public, but the parameters are private)

General setup

- A server holds a *database* D consisting of (equal-sized, padded if necessary) *records*
  - Say there are r records, each of size s
- A client has a query q
  - A record number, or a keyword, for example
- Desired outcome: client learns the record corresponding to q, server learns nothing about q
  - It's usually OK if the client happens to learn *more* information about D as well, but sometimes not

A trivial solution

- Here is a trivial protocol to achieve this:
- Client sends to server: "I would like to make a query"
- Server sends to client: the whole database D
- Client looks up the information in the database themselves
- Pro: very simple ("trivial")
  Con: communication the size of D (which is $r \cdot s$)

Communicating less data

- We want "true" PIR solutions to communicate less data than the whole database, while still not revealing anything about the query
  - Asking for just half of the database, for example, reveals that the query was in that half, so that's no good
- You can take any of our three private computation approaches to solve this problem:
  - Distributed trust
  - Trusted hardware
  - Homomorphic encryption
- We'll look at the distributed trust solution now

Multi-server PIR

- In the (simplest version of the) distributed trust setting, there are *multiple* servers, *each* with a copy of the database D
- The client secret shares the query q and sends one share to each server
- Each server processes its share of q to produce a share of the desired response, which it returns to the client
- The client combines the response shares to get the complete response

The database as a matrix

- Most PIR protocols will model the database D as a matrix
  - For example, a matrix with r rows, each of length s bytes
  - The i^{th} row of the matrix is the i^{th} record of the database

$$D = \begin{bmatrix} \text{Sealing assembly for } \ldots \\ \text{Adjustable-backset } \ldots \\ \text{Conical recreational } \ldots \\ \text{Method of swinging } \ldots \\ \text{Cover for the rails } \ldots \\ \text{Golf ball delivery } \ldots \end{bmatrix}$$

- If you write your query like this: $q = [\ 0\ 0\ 0\ 1\ 0\ 0\ ]$ then what is $q \cdot D$?

A simple PIR protocol

- A very simple PIR protocol (from the original PIR paper due to Chor et al.):
- n servers each have a copy of D
- The client writes their query q as e_i (a vector of all 0s except a 1 in position i)
- The client XOR-shares q into n shares to get $q_1, \ldots, q_n$ where $q_1 \oplus \cdots \oplus q_n = q$, sends q_j to server j for each $j = 1, \ldots, n$

A simple PIR protocol

- Server j computes its answer $a_j = q_j \cdot D$
  - q_j will be a vector of length r of random bits (0 or 1)
  - $a_j = q_j \cdot D$ is just saying "for each index i where the i^{th} entry of q_j is 1, XOR those records of D together to get a_j"
- Server j sends a_j back to the client
- The client computes $a = a_1 \oplus \cdots \oplus a_n$
- How much data is transmitted?
  - q_j has length r bits, a_j has length s bytes, there are n servers, so the client sends nr bits and receives ns bytes
  - $n\lceil \frac{r}{8} \rceil + ns$ is (likely) a *lot* smaller than rs (the size of the whole database)

Properties of this protocol

- Expressibility: (index) PIR
- Minimum number of parties: $n \geq 2$ servers
- Threat model: semi-honest
- Maximum number of adversarial parties: $n - 1$
- Performance (r records of size s):
  - Local computation: $\mathcal{O}(n(r + s))$ client, $\mathcal{O}(rs)$ per server
  - Total communication: $n(\lceil \frac{r}{8} \rceil + s)$
  - Latencies: 2

Extensions

- There are *many* ways to extend and improve this simple PIR protocol
- Some examples:
  - Batching (reducing computation)
  - Threat model
  - Robustness
  - Reducing communication

Reducing computation with batching

- To answer a query, the servers have to do *some* computation over the entire database
  - If they ignore some record, then that record was definitely not the query
- But it turns out to answer *lots* of queries (say m) at the same time, the servers can do $o(mrs)$ work
  - We assume m is much smaller than r and s
- Two cases:
  - A *single* client making lots of queries
  - Lots of clients making one query each

Batch codes

- In the first case, you have a single client who wants to look up a lot of queries at the same time
- We won't go into the details here, but one technique is *batch codes*
- Rather than encoding the queries as $q = [\ 0\ 0\ 0\ 1\ 0\ 0\ ]$ for example, the client uses better encodings
- In one variant, for example, the servers only have to do $\mathcal{O}(m^{0.415} rs)$ work
  - But the response size is much larger, at $m^2 s$ (instead of ms)

Independent clients

- Batch codes only work if a single client can encode lots of queries in a clever manner
- If you have lots of independent clients, they're each going to submit their query as if they were the only one
- But the server can still save computation!

Independent clients

- Recall that each server j is computing $a_j = q_j \cdot D$
- If m queries $q_j^{(1)}, \ldots, q_j^{(m)}$ come in at the same time, *stack* them into a matrix Q_j
  - Each row of Q_j is one of the queries

$$Q_j = \begin{bmatrix} \text{---} & q_j^{(1)} & \text{---} \\ \text{---} & q_j^{(2)} & \text{---} \\ & \vdots & \\ \text{---} & q_j^{(m)} & \text{---} \end{bmatrix}$$

Independent clients

- Recall that each server j is computing $a_j = q_j \cdot D$
- If m queries $q_j^{(1)}, \ldots, q_j^{(m)}$ come in at the same time, *stack* them into a matrix Q_j
  - Each row of Q_j is one of the queries

$$Q_j \cdot D = \begin{bmatrix} \text{———} & q_j^{(1)} & \text{———} \\ \text{———} & q_j^{(2)} & \text{———} \\ & \vdots & \\ \text{———} & q_j^{(m)} & \text{———} \end{bmatrix} \cdot D$$

Independent clients

- Recall that each server j is computing $a_j = q_j \cdot D$
- If m queries $q_j^{(1)}, \ldots, q_j^{(m)}$ come in at the same time, *stack* them into a matrix Q_j
  - Each row of Q_j is one of the queries

$$Q_j \cdot D = \begin{bmatrix} \text{———} & q_j^{(1)} & \text{———} \\ \text{———} & q_j^{(2)} & \text{———} \\ & \vdots & \\ \text{———} & q_j^{(m)} & \text{———} \end{bmatrix} \cdot D = \begin{bmatrix} \text{————} & a_j^{(1)} & \text{————} \\ \text{————} & a_j^{(2)} & \text{————} \\ & \vdots & \\ \text{————} & a_j^{(m)} & \text{————} \end{bmatrix}$$

Independent clients

- It takes $\mathcal{O}(rs)$ work to multiply a $1 \times r$ vector by an $r \times s$ matrix
- But you can multiply an $m \times r$ matrix by an $r \times s$ matrix in less than m times that cost
- $\mathcal{O}(m^{0.81} rs)$ is easy, lower numbers are theoretically possible
- Also: no expansion of response size

Threat model and robustness

- The presented protocol used XOR sharing
- Excellent resistance to collusion (up to $n-1$), but the protocol completely fails if even one server refuses to answer, or (intentionally) gives an incorrect response
- You can fix this by using different secret sharing
  - e.g., t-of-n Shamir secret sharing
  - Then you can handle both servers that fail to respond and malicious servers that give incorrect responses
  - But the resistance to collusion goes down to $t-1$

Reducing communication

- Another way to improve this protocol is to reduce the amount of *communication*
  - Query size or response size or both
  - Sometimes this increases the computation cost, so there's a tradeoff

- Recall the (non-private) query $q = [\ 0\ 0\ 0\ 1\ 0\ 0\ ]$

- One can consider $q(i)$ (the i^{th} element of q) to be a "point function": a function that's 0 everywhere except in one position
  - Since q is a bit vector, that position necessarily is a 1

Point functions

- A *point function* is a function that is non-zero at exactly one input:

$$p_{a,b}(i) = \begin{cases} 0 & i \neq a \\ b & i = a \end{cases}$$

- For a *binary* point function, the outputs are all either 0 or 1, so b must be 1

- For a general point function, b can be any (non-zero) valid output

Distributed point functions

- An (n, t)-*distributed point function* (DPF) is a way to construct n secret shares of a point function so that:
  - Any t shares can be used to reconstruct the original point function $p_{a,b}$
  - Any $t-1$ shares *cannot* be used to learn a or b (unless you know $b=1$ because it's a binary DPF)
- One way to do it we've already seen: write the point function as a vector of its outputs $q = [\ 0\ 0\ 0\ 1\ 0\ 0\ ]$ and secret share that vector
  - But the problem we wanted to address is that, if there are r possible inputs, this vector (and its shares) is of length r, which could be very large

(2,2)-DPFs

- We're going to look at the simplest case: (2,2)-DPFs
  - There are two shares, and neither share alone can reveal a (or b if not binary)

- API: $\textsc{Gen}(r, a, b) \to (key_0, key_1)$
  - Given the size of the set of possible inputs r, a target input a (with $0 \leq a < r$) and a target output b, produce a pair of *DPF keys*. Send key_β to server β for $\beta \in \{0, 1\}$
  - Note: we will want the sizes of key_0 and key_1 to be smaller than r

- API: $\textsc{Eval}(\beta, key_\beta, i) \to v_\beta^i$
  - Server β uses key_β to evaluate its share of the DPF at input i, yielding v_β^i, which should reveal nothing about a or b

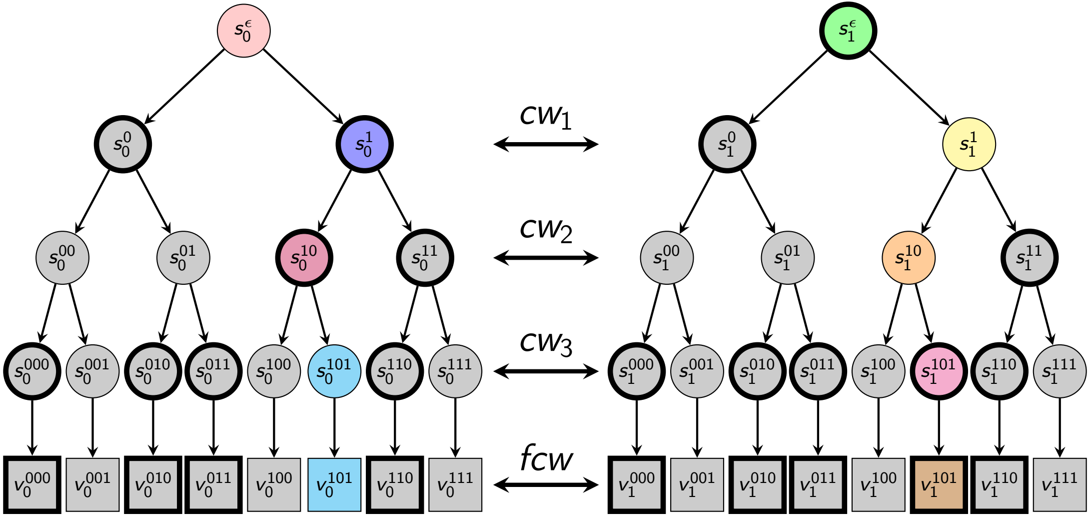
(2,2)-DPFs

API: $\textsc{Gen}(r, a, b) \rightarrow (key_0, key_1)$
API: $\textsc{Eval}(\beta, key_\beta, i) \rightarrow v^i_\beta$

- Property: for each i, $v^i_0 \oplus v^i_1 = p_{a,b}(i)$
- That is, for $i \neq a$, $v^i_0 = v^i_1$, and for $i = a$, $v^i_0 \oplus v^i_1 = b$

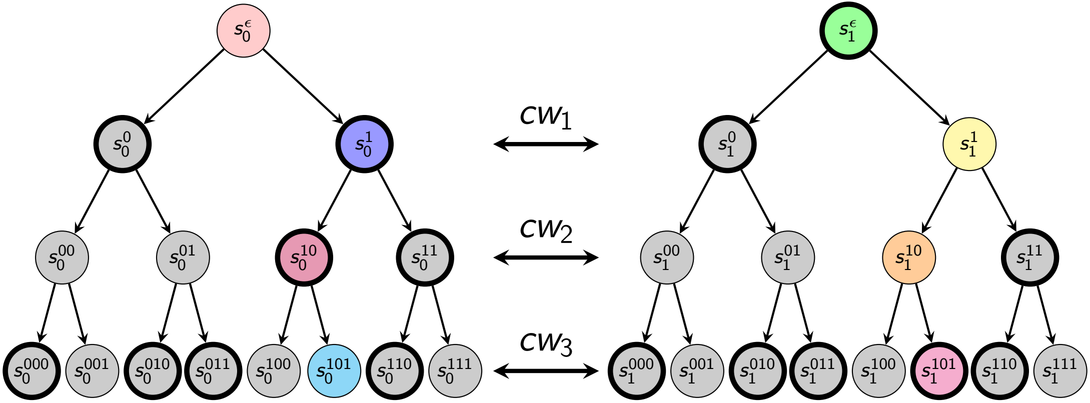
- How do we implement $\textsc{Gen}$ and $\textsc{Eval}$?
- Strategy: *visualize* all possible inputs i to $\textsc{Eval}$ as a binary tree
  - Note: you won't actually *construct* this binary tree at any point!

(2,2)-DPFs



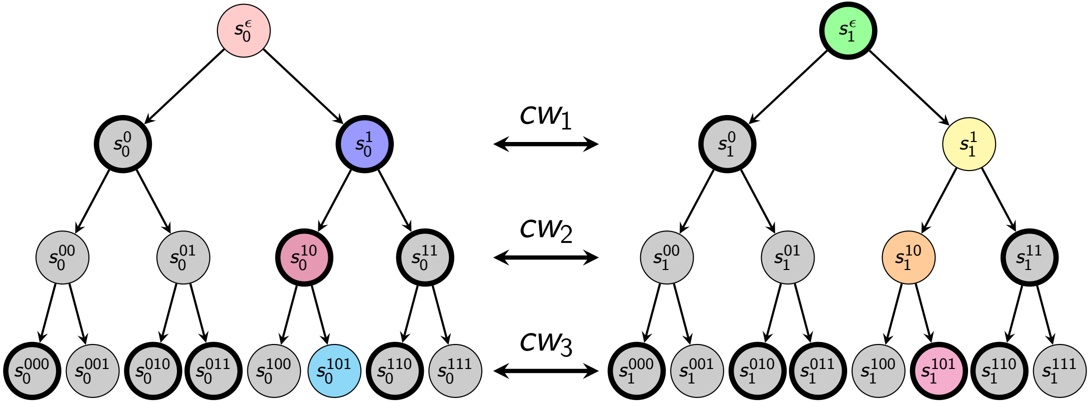
s_0^ϵ
s_1^ϵ
s_0^0 s_0^1
s_1^0 s_1^1
cw_1
s_0^{00} s_0^{01} s_0^{10} s_0^{11}
s_1^{00} s_1^{01} s_1^{10} s_1^{11}
cw_2
s_0^{000} s_0^{001} s_0^{010} s_0^{011} s_0^{100} s_0^{101} s_0^{110} s_0^{111}
s_1^{000} s_1^{001} s_1^{010} s_1^{011} s_1^{100} s_1^{101} s_1^{110} s_1^{111}
cw_3
v_0^{000} v_0^{001} v_0^{010} v_0^{011} v_0^{100} v_0^{101} v_0^{110} v_0^{111}
v_1^{000} v_1^{001} v_1^{010} v_1^{011} v_1^{100} v_1^{101} v_1^{110} v_1^{111}
fcw

(2,2)-DPFs



s_0^ϵ
s_0^0
s_0^1
s_0^{00}
s_0^{01}
s_0^{10}
s_0^{11}
s_0^{000}
s_0^{001}
s_0^{010}
s_0^{011}
s_0^{100}
s_0^{101}
s_0^{110}
s_0^{111}
CW_1
CW_2
CW_3
s_1^ϵ
s_1^0
s_1^1
s_1^{00}
s_1^{01}
s_1^{10}
s_1^{11}
s_1^{000}
s_1^{001}
s_1^{010}
s_1^{011}
s_1^{100}
s_1^{101}
s_1^{110}
s_1^{111}

(2,2)-DPFs



- $key_0 = (s_0 = s_0^\epsilon, cw_1, cw_2, cw_3)$ $\quad key_1 = (s_1 = s_1^\epsilon, cw_1, cw_2, cw_3)$
- Each $cw_k = (sc_k, fc_k^0, fc_k^1)$

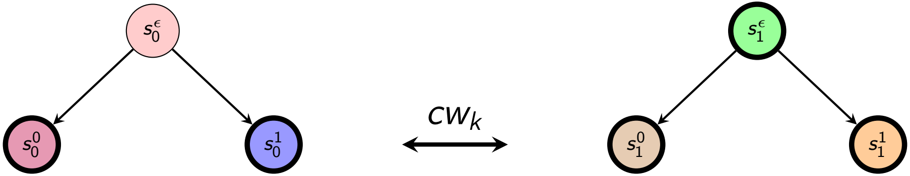
DPF nodes

- Each node in the (again, notional) DPF tree has:
  - A *seed* (typically around 128 bits)
  - A *flag bit* (one bit)
- We will denote the seed for server β at the node with prefix α by s_β^α
- We will denote the flag bit for a node by a thick outline if the flag bit is 1, and a thin outline if it is 0

Children of DPF nodes

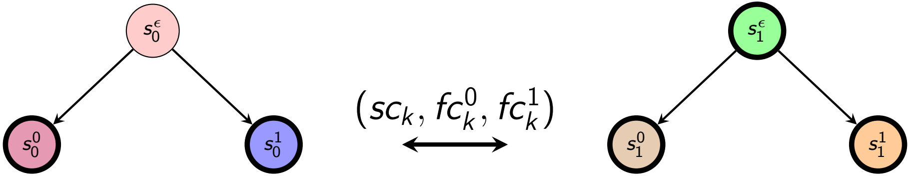
- To get the seeds and flag bits for the children of a given parent node:
  - Use the seed of the parent node as the input to a PRG. Treat the output of the PRG as (left seed, left flag, right seed, right flag); these will all be random values

Children of DPF nodes



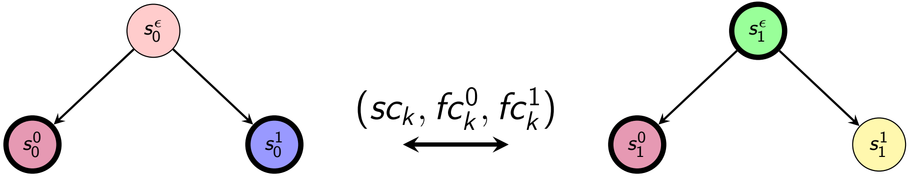
- To get the seeds and flag bits for the children of a given parent node:
  - Use the seed of the parent node as the input to a PRG. Treat the output of the PRG as (left seed, left flag, right seed, right flag); these will all be random values
  - **If the parent's flag bit is 1**: XOR sc_k into both children's seeds, XOR fc_k^0 into the left child's flag bit, XOR fc_k^1 into the right child's flag bit

Children of DPF nodes



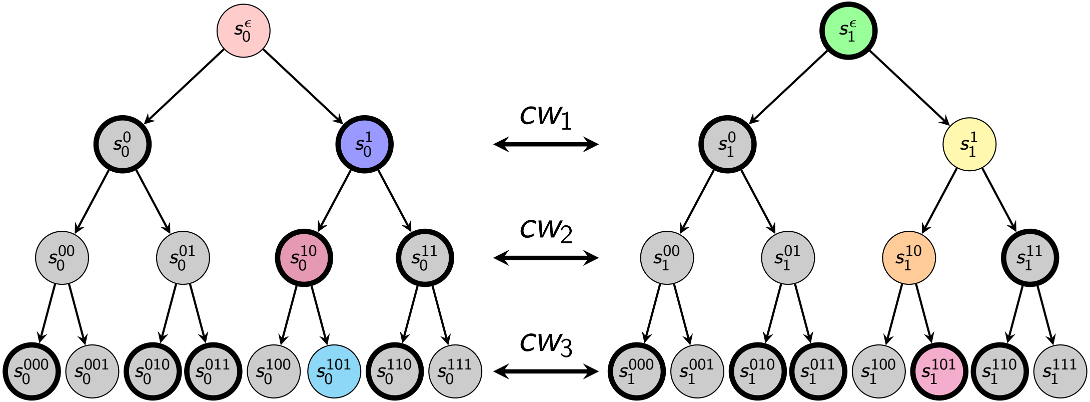
- To get the seeds and flag bits for the children of a given parent node:
  - Use the seed of the parent node as the input to a PRG. Treat the output of the PRG as (left seed, left flag, right seed, right flag); these will all be random values
  - **If the parent's flag bit is 1**: XOR sc_k into both children's seeds, XOR fc_k^0 into the left child's flag bit, XOR fc_k^1 into the right child's flag bit

Children of DPF nodes



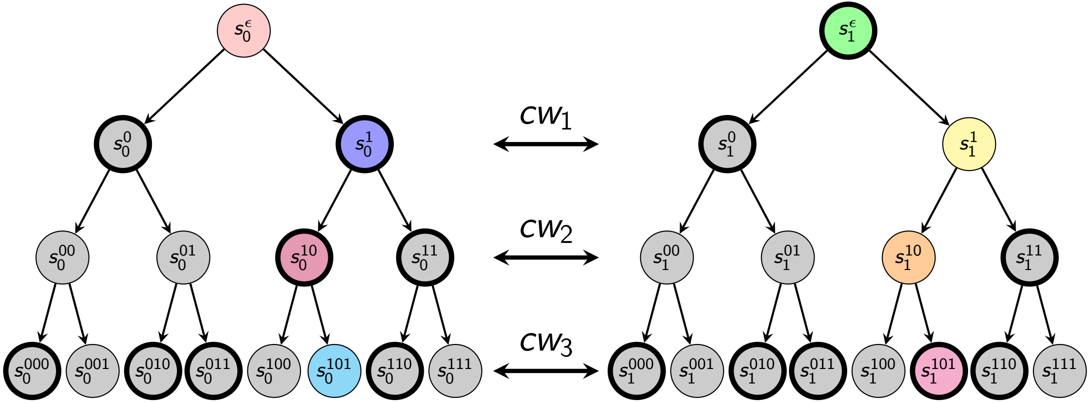
- To get the seeds and flag bits for the children of a given parent node:
  - Use the seed of the parent node as the input to a PRG. Treat the output of the PRG as (left seed, left flag, right seed, right flag); these will all be random values
  - **If the parent's flag bit is 1**: XOR sc_k into both children's seeds, XOR fc_k^0 into the left child's flag bit, XOR fc_k^1 into the right child's flag bit
  - In this case, $sc_k = PRG(s_0^\epsilon)[\text{left seed}] \oplus PRG(s_1^\epsilon)[\text{left seed}]$,
    $fc_k^0 = PRG(s_0^\epsilon)[\text{left flag}] \oplus PRG(s_1^\epsilon)[\text{left flag}]$,
    $fc_k^1 = PRG(s_0^\epsilon)[\text{right flag}] \oplus PRG(s_1^\epsilon)[\text{right flag}] \oplus 1$

The DPF trees



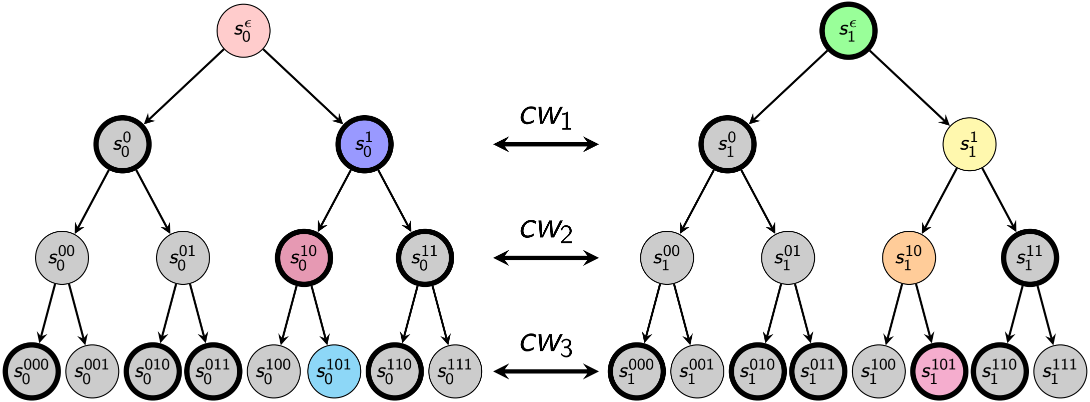
- Invariant: each node on the path leading to the target index a has a *different* seed and a *different* flag in the two trees; each node not on this path has the *same* seed and flag in the two trees

The DPF trees



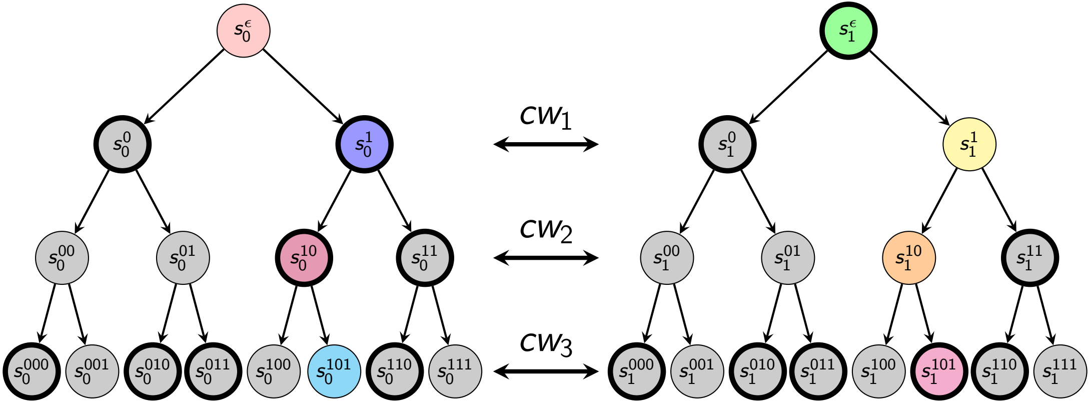
- For a binary DPF, we're done: look at the flag bits at the leaves; they are identical except for the target index
- So $\textsc{Eval}(\beta, key_\beta, i)$ is just the flag bit at leaf i

The DPF trees



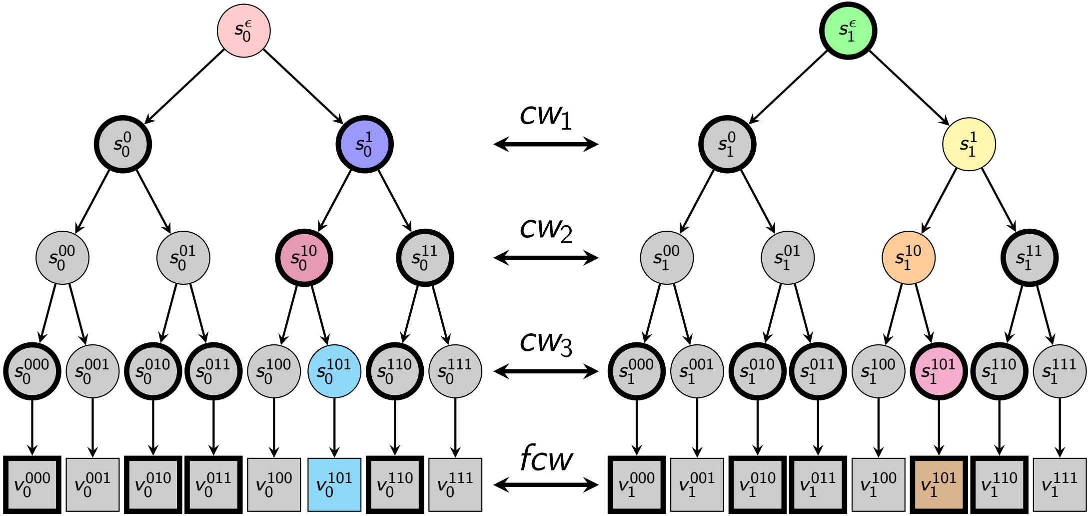
- And remember, when computing $\textsc{Eval}(\beta, key_\beta, i)$, you *only* compute the seeds and flags on the path from the root to i, and not any others

The DPF trees



- For non-binary DPFs, two extra steps: first, hash the seed you end up with into however large an output you need, then, if the flag bit is 1, XOR that with a *final correction word*

Non-binary DPF trees



Properties of this protocol

- Expressibility: (index) PIR
- Minimum number of parties: 2 servers
- Threat model: semi-honest
- Maximum number of adversarial parties: 1
- Performance (r records of size s):
    - Local computation: $\mathcal{O}(s + \lg r)$ client, $\mathcal{O}(rs)$ per server
    - Total communication: [Assignment 2]
    - Latencies: 2

Keyword PIR

- Up to now, we have assumed that the client knows the exact database index of the record they're looking for
  - For something like patent numbers, where the number could itself just be the index, that might be OK
- But in general, a (keyword, value) store is much more useful
  - Sometimes called a (key, value) store, but "key" of course already has a different meaning in privacy / cryptography
- The database is a collection of (keyword, value) pairs
- The client has a keyword, and wants to look up the associated value **without revealing the keyword**
  - Or be told that no such value exists

Keyword PIR

- One technique is to put the values in an index-PIR database (as before), and then have a separate mechanism (which could be based on PIR accesses into a binary search tree, for example) to look up the correct index for a given keyword
- This will require multiple communication rounds and additional computation, however
- Using DPFs, we can achieve keyword PIR with almost the same performance as index PIR

The two hashes

- For each (keyword, value) pair in the database, hash the keyword in two ways:
  - A regular hash; e.g., SHA2-256 with a 32-byte output
  - A *truncated* hash which is the first d bits of the regular hash
- d is chosen so that no two keywords have the same truncated hash
  - If the keywords in the database can be chosen adversarially, choose $d = 256$ (i.e., use the whole hash, not truncated)
  - Otherwise, choosing $d = 2\lceil \lg r \rceil$ (where r is the number of keywords in the database) is typically fine
- Notation: for a keyword w, $H(w)$ will be the full hash, $H_d(w)$ will be the hash truncated to the first d bits

One more notation

- For any (keyword, value) pair (w, v) in the database, let

$$V(w) = H(w) \| v$$

- That is, $V(w)$ is (the 32-byte hash of the *keyword*) concatenated with (the *value*)
- So if values are s bytes long, $V(w)$ will be $32 + s$ bytes long

Converting DPF-based index PIR to keyword PIR

Client

Server β

$$\textsc{Gen}(r, i, 1) \rightarrow (key_0, key_1)$$

$$a_\beta = \bigoplus_{\substack{j \in \{0,\ldots,r-1\} \\ \textsc{Eval}(\beta, key_\beta, j) = 1}} D[j]$$

$$a = a_0 \oplus a_1$$

Converting DPF-based index PIR to keyword PIR

Client

Server β

$$\textsc{Gen}(2^d, H_d(w), 1) \rightarrow (key_0, key_1)$$

$$a_\beta = \bigoplus_{\substack{j \in \{0,\ldots,r-1\} \\ \textsc{Eval}(\beta, key_\beta, j)=1}} D[j]$$

$$a = a_0 \oplus a_1$$

Converting DPF-based index PIR to keyword PIR

Client

Server β

$\textsc{Gen}(2^d, H_d(w), 1) \rightarrow (key_0, key_1)$

$$a_\beta = \bigoplus_{\substack{w \in \text{keywords} \\ \textsc{Eval}(\beta, key_\beta, H_d(w)) = 1}} V(w)$$

$a = a_0 \oplus a_1$

Converting DPF-based index PIR to keyword PIR

Client

Server β

$\text{Gen}(2^d, H_d(w), 1) \rightarrow (key_0, key_1)$

$$a_\beta = \bigoplus_{\substack{w \in \text{keywords} \\ \text{Eval}(\beta, key_\beta, H_d(w)) = 1}} V(w)$$

$a = a_0 \oplus a_1$

Check a starts with $H(w)$

Properties of this protocol

- Expressibility: keyword PIR
- Minimum number of parties: 2 servers
- Threat model: semi-honest
- Maximum number of adversarial parties: 1
- Performance (r records of size s):
  - Local computation: $\mathcal{O}(s + \lg r)$ client, $\mathcal{O}(rs)$ per server
  - Total communication: [Assignment 2]
  - Latencies: 2

Private Set Intersection (PSI)

- Another multiparty protocol to compute a specific function is *private set intersection* (PSI)

- In its simplest form, there are two parties, the *receiver* and the *sender*

- Each party has a set of *elements*
  - Numbers, strings, IP addresses, whatever

- The goal is for the receiver to learn which elements the two parties have in common
  - Both parties can learn (a bound on) the size of each other's sets
  - The sender learns nothing else

Uses of PSI

- Google and Mastercard: what users bought something they saw in a Google ad?
- Messaging apps: which of your friends are already users of this app?
- Contact tracing: what places I have visited have had a reported COVID exposure?

Variants

- PSI Cardinality
  - The receiver only learns the *number* of items in common
  - More generally, compute some function of the intersection
- Unbalanced PSI: the sender or receiver has a much larger set than the other
  - Large sender set: messaging app example
  - Large receiver set: contact tracing example
- Private Set Union (Cardinality)
  - Find the (number of) users a set of services have in total, without double-counting people that use multiple services

Comparison of PIR and PSI

- If the receiver has only one element, and the sender has a database of elements, PSI is a little bit like keyword PIR
- But in keyword PIR, the client *is* allowed to learn information about other entries in the database, and in PSI, the receiver is *not*
  - Symmetric PIR (SPIR)
- The database in PSI is held by one party
  - The PIR protocols we've seen so far require at least two (non-colluding) parties to hold copies of the database
  - But we'll see single-party PIR protocols in future modules

A simple but broken PSI protocol

- Let the sender's set be $S = \{s_1, s_2, \ldots, s_m\}$ and the receiver's set be $R = \{r_1, r_2, \ldots, r_n\}$
- The sender computes hashes of its elements $H(s_1), H(s_2), \ldots, H(s_m)$ and sends them to the receiver
- The receiver hashes its own elements and looks for matches
- Why is this insecure?

A simple PSI protocol

- The sender hashes their elements to points in a group: $P_1 = H_p(s_1), P_2 = H_p(s_2), \ldots, P_m = H_p(s_m)$
- The receiver does the same: $Q_1 = H_p(r_1), Q_2 = H_p(r_2), \ldots, Q_n = H_p(r_n)$
- The receiver picks a random scalar a and sends to the sender: $a \cdot Q_1, a \cdot Q_2, \ldots, a \cdot Q_n$
- The sender picks a random scalar b and sends to the receiver: $b \cdot P_1, b \cdot P_2, \ldots, b \cdot P_m$ and $H(ba \cdot Q_1), H(ba \cdot Q_2), \ldots, H(ba \cdot Q_n)$
- The receiver computes $H(ab \cdot P_1), H(ab \cdot P_2), \ldots, H(ab \cdot P_m)$ and finds the values in common

A simple PSI protocol

- The sender hashes their elements to points in a group: $P_1 = H_p(s_1), P_2 = H_p(s_2), \ldots, P_m = H_p(s_m)$
- The receiver does the same: $Q_1 = H_p(r_1), Q_2 = H_p(r_2), \ldots, Q_n = H_p(r_n)$
- The receiver picks a random scalar a and sends to the sender: $a \cdot Q_1, a \cdot Q_2, \ldots, a \cdot Q_n$
- The sender picks a random scalar b and sends to the receiver: $b \cdot P_1, b \cdot P_2, \ldots, b \cdot P_m$ and $H(ba \cdot Q_1), H(ba \cdot Q_2), \ldots, H(ba \cdot Q_n)$
- The receiver computes $H(ab \cdot P_1), H(ab \cdot P_2), \ldots, H(ab \cdot P_m)$ and finds the values in common
- Why do we not have the same problem as before?

Properties of this protocol

- Expressibility: balanced PSI
- Minimum number of parties: 2 servers
- Threat model: semi-honest
- Maximum number of adversarial parties: 1
- Performance (sender has m elements, receiver has n):
  - Local computation: $\mathcal{O}(m + n)$
  - Total communication: 32m+64n bytes
  - Latencies: 2

Secret sharing without reconstruction

- In Module 2, we saw how to share a secret (say a private key) using Shamir secret sharing
  - Prevents the secret from sitting on a single computer, which would then be vulnerable
- We also saw how to reconstruct the secret using Lagrange interpolation so that it can be used (say to sign a message)
  - But once the secret is reconstructed, it's vulnerable again!
- Better: be able to use the shared private key to sign a message *without* reconstructing it!
  - Key idea: use shares of the key to produce shares of the signature, and only reconstruct the signature, not the key

Schnorr signatures

m, a

$A = a \cdot B$

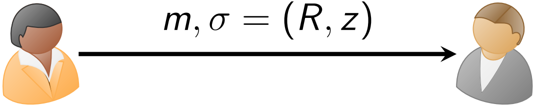
Schnorr signatures

m, a

$A = a \cdot B$

$$r \leftarrow \$$$
$$R \leftarrow r \cdot B$$
$$c \leftarrow H(R, A, m)$$
$$z \leftarrow r + c \cdot a$$

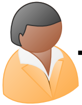
Schnorr signatures



$$r \leftarrow \$$$
$$R \leftarrow r \cdot B$$
$$c \leftarrow H(R, A, m)$$
$$z \leftarrow r + c \cdot a$$

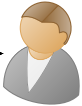
Schnorr signatures

m, a



$$m, \sigma = (R, z) \longrightarrow$$

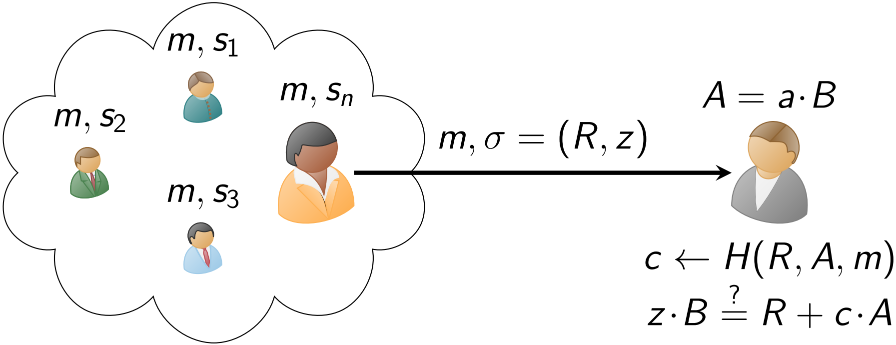
$A = a \cdot B$



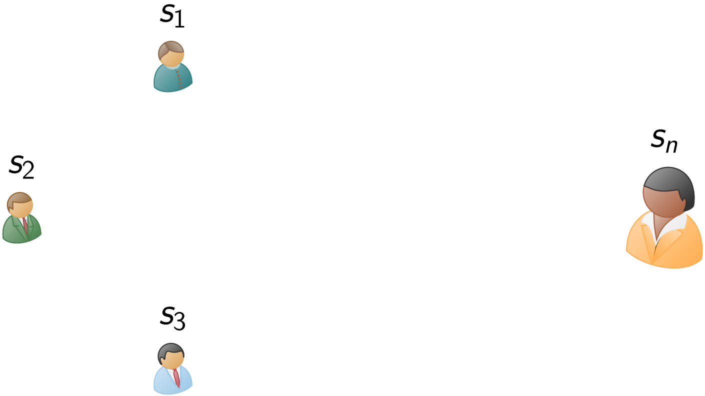
$$r \leftarrow \$$$
$$R \leftarrow r \cdot B$$
$$c \leftarrow H(R, A, m)$$
$$z \leftarrow r + c \cdot a$$

$$c \leftarrow H(R, A, m)$$
$$z \cdot B \stackrel{?}{=} R + c \cdot A$$

Threshold Schnorr signatures



Two-Round threshold Schnorr signatures



Two-Round threshold Schnorr signatures

s_1

$r_1 \leftarrow \$$

$R_1 \leftarrow r_1 \cdot B$

s_2

$r_2 \leftarrow \$$

$R_2 \leftarrow r_2 \cdot B$

s_3

$r_3 \leftarrow \$$

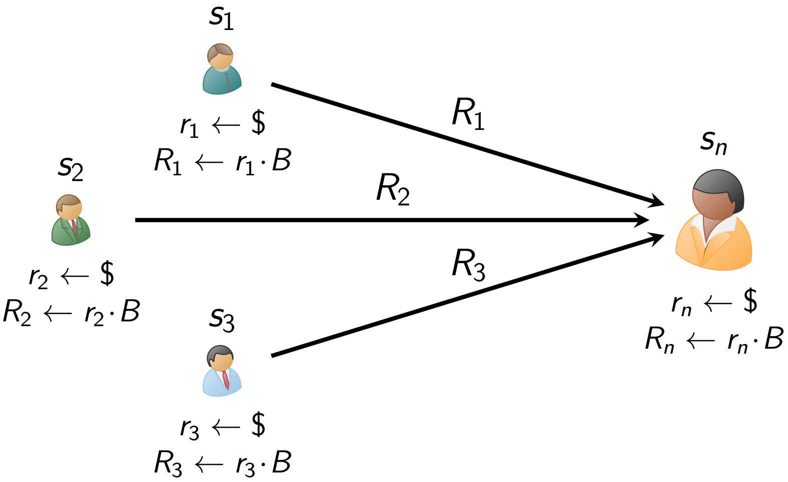
$R_3 \leftarrow r_3 \cdot B$

s_n

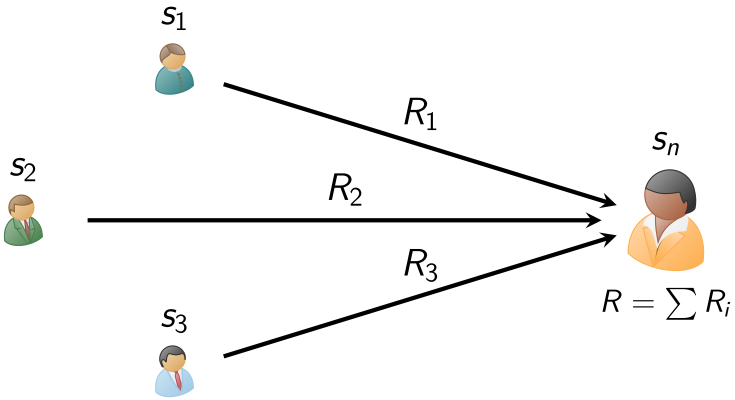
$r_n \leftarrow \$$

$R_n \leftarrow r_n \cdot B$

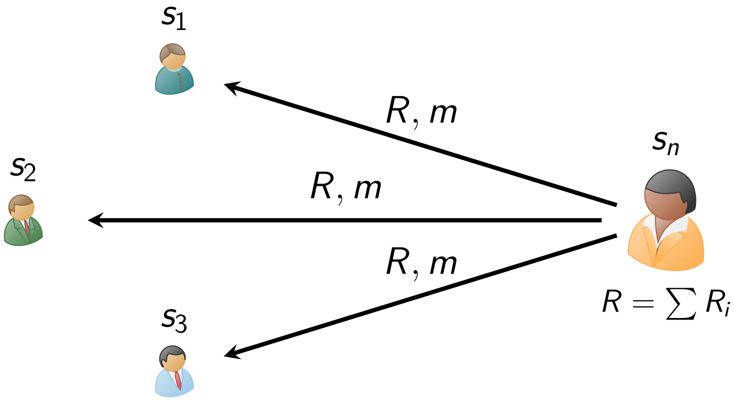
Two-Round threshold Schnorr signatures



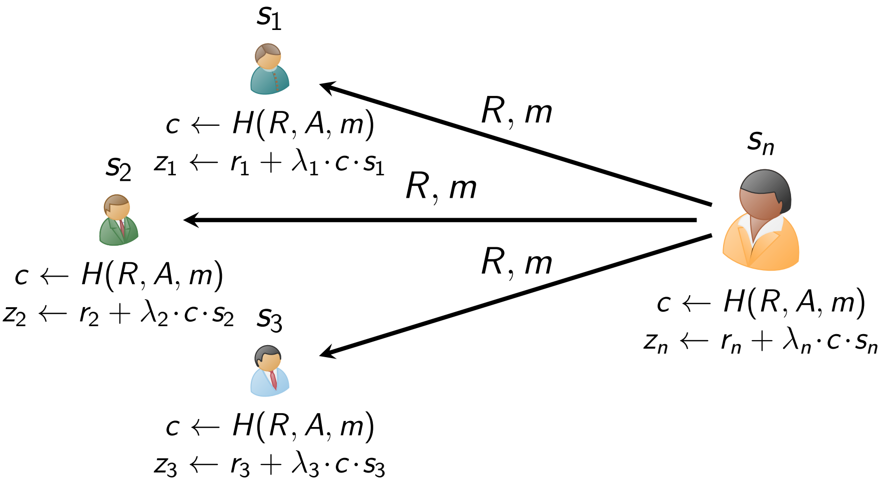
Two-Round threshold Schnorr signatures



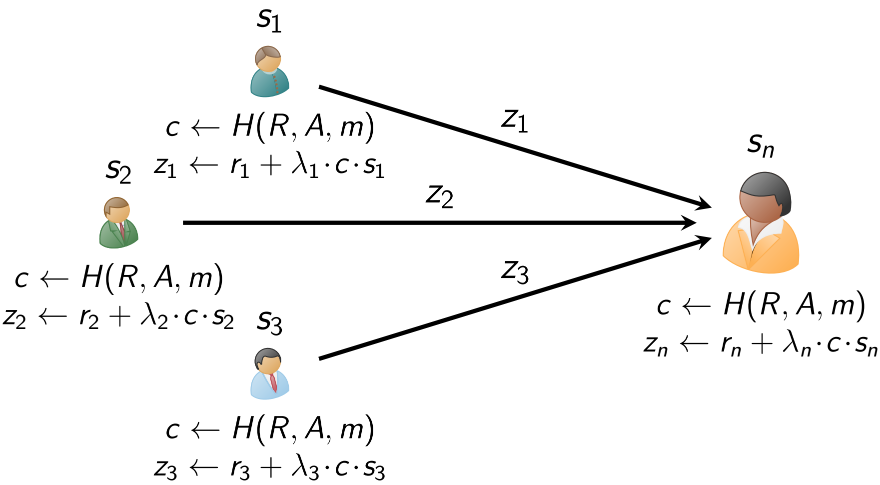
Two-Round threshold Schnorr signatures



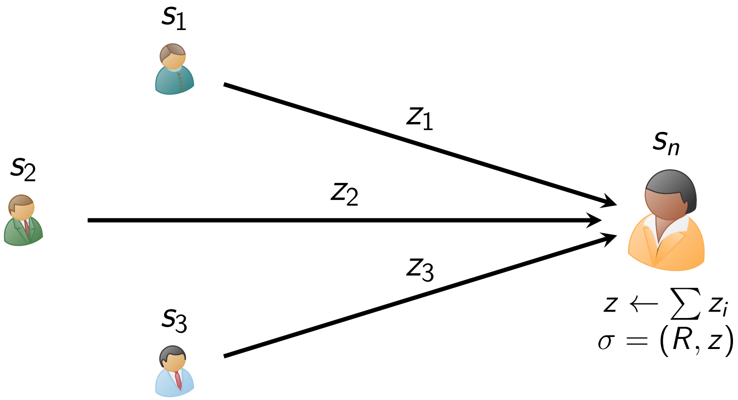
Two-Round threshold Schnorr signatures



Two-Round threshold Schnorr signatures



Two-Round threshold Schnorr signatures



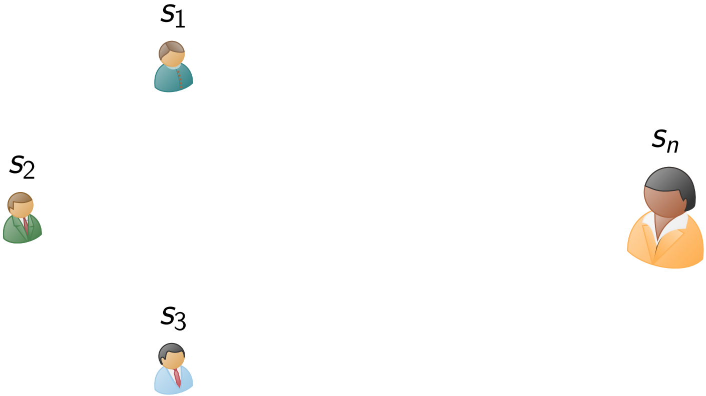
Problem: parallel composition

2019 IEEE Symposium on Security and Privacy

On the Security of Two-Round Multi-Signatures

Manu Drijvers[*†], Kasra Edalatnejad[‡], Bryan Ford[‡], Eike Kiltz[§], Julian Loss[§], Gregory Neven[*], Igors Stepanovs[¶]
[*]DFINITY, [†]ETH Zurich, [‡]EPFL, [§]Ruhr-Universität Bochum, [¶]UCSD.

FROST



FROST

s_1

$d_1, e_1 \leftarrow \$$

$D_1 \leftarrow d_1 \cdot B$

$E_1 \leftarrow e_1 \cdot B$

s_2

$d_2, e_2 \leftarrow \$$

$D_2 \leftarrow d_2 \cdot B$

$E_2 \leftarrow e_2 \cdot B$

s_3

$d_3, e_3 \leftarrow \$$

$D_3 \leftarrow d_3 \cdot B$

$E_3 \leftarrow e_3 \cdot B$

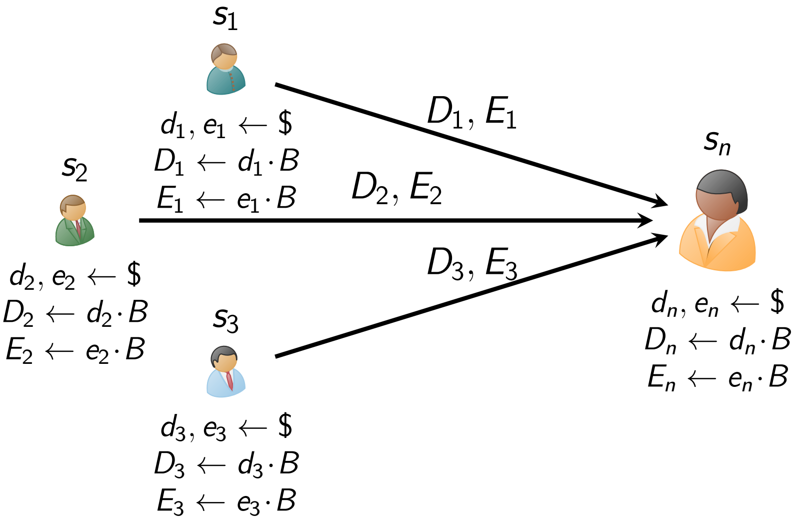
s_n

$d_n, e_n \leftarrow \$$

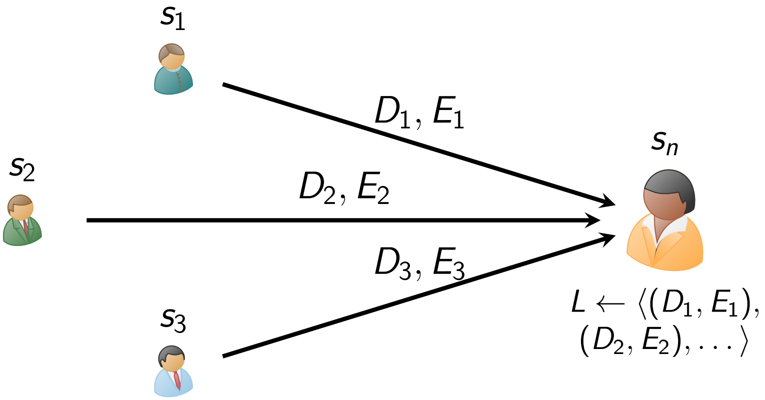
$D_n \leftarrow d_n \cdot B$

$E_n \leftarrow e_n \cdot B$

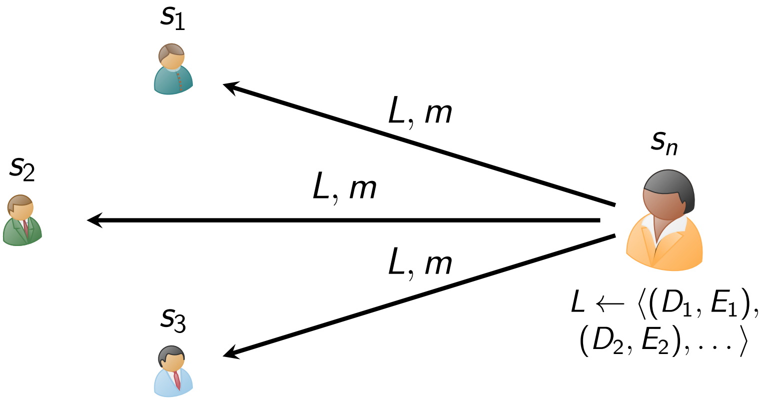
FROST


s_1
$d_1, e_1 \leftarrow \$$
$D_1 \leftarrow d_1 \cdot B$
$E_1 \leftarrow e_1 \cdot B$
s_2
$d_2, e_2 \leftarrow \$$
$D_2 \leftarrow d_2 \cdot B$
$E_2 \leftarrow e_2 \cdot B$
s_3
$d_3, e_3 \leftarrow \$$
$D_3 \leftarrow d_3 \cdot B$
$E_3 \leftarrow e_3 \cdot B$
s_n
$d_n, e_n \leftarrow \$$
$D_n \leftarrow d_n \cdot B$
$E_n \leftarrow e_n \cdot B$
D_1, E_1
D_2, E_2
D_3, E_3

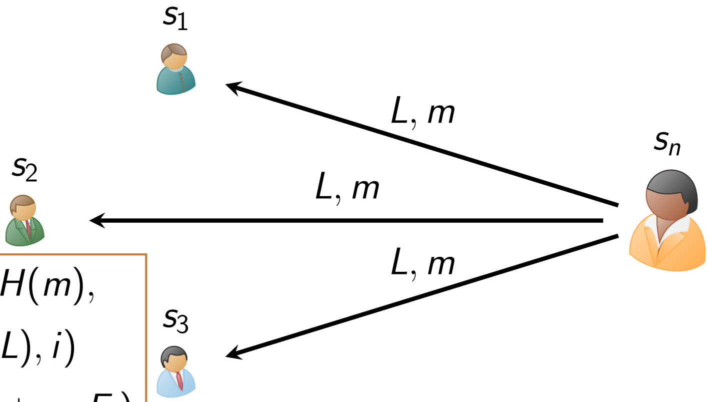
FROST


s_1
s_2
s_3
s_n
D_1, E_1
D_2, E_2
D_3, E_3
$L \leftarrow \langle (D_1, E_1),$
$(D_2, E_2), \ldots \rangle$

FROST



FROST

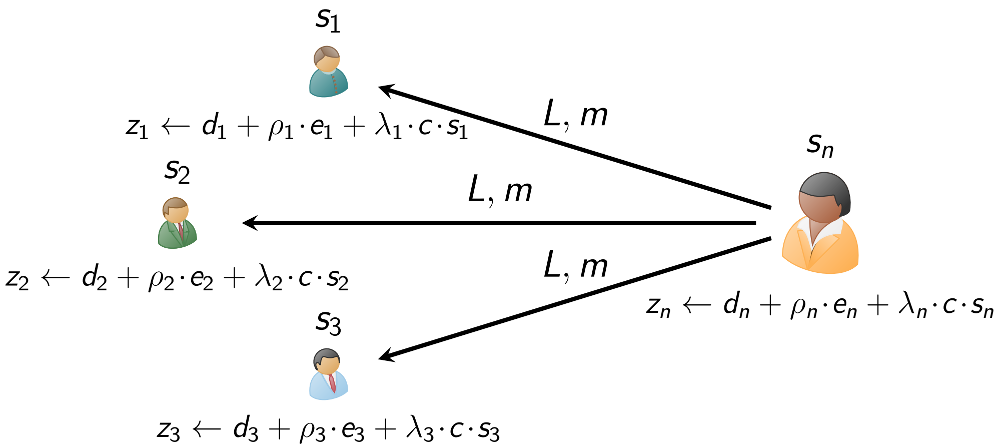


$$\rho_i \leftarrow H_\rho(A, H(m), H(L), i)$$

$$R \leftarrow \sum (D_i + \rho_i \cdot E_i)$$

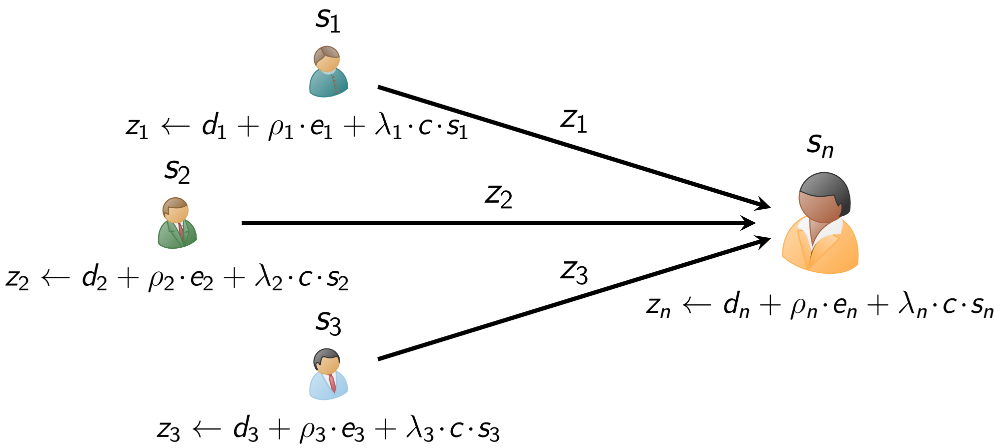
$$c \leftarrow H(R, A, m)$$

FROST



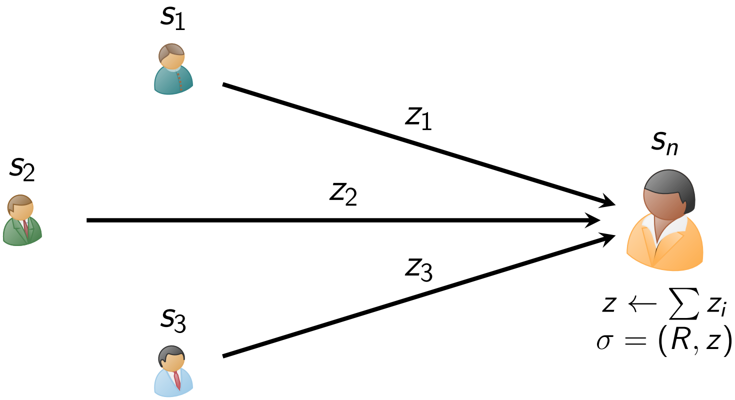
s_1
$z_1 \leftarrow d_1 + \rho_1 \cdot e_1 + \lambda_1 \cdot c \cdot s_1$
L, m
s_2
L, m
$z_2 \leftarrow d_2 + \rho_2 \cdot e_2 + \lambda_2 \cdot c \cdot s_2$
L, m
s_n
$z_n \leftarrow d_n + \rho_n \cdot e_n + \lambda_n \cdot c \cdot s_n$
s_3
$z_3 \leftarrow d_3 + \rho_3 \cdot e_3 + \lambda_3 \cdot c \cdot s_3$

FROST



s_1
$z_1 \leftarrow d_1 + \rho_1 \cdot e_1 + \lambda_1 \cdot c \cdot s_1$
s_2
$z_2 \leftarrow d_2 + \rho_2 \cdot e_2 + \lambda_2 \cdot c \cdot s_2$
s_3
$z_3 \leftarrow d_3 + \rho_3 \cdot e_3 + \lambda_3 \cdot c \cdot s_3$
z_1
z_2
z_3
s_n
$z_n \leftarrow d_n + \rho_n \cdot e_n + \lambda_n \cdot c \cdot s_n$

FROST



Properties of this protocol

- Expressibility: threshold Schnorr signatures
- Minimum number of parties: $n \geq t$
- Threat model: malicious
- Maximum number of adversarial parties: $t - 1$
- Performance:
  - Local computation: $\mathcal{O}(t + |m|)$ per party
  - Total communication: $64t$ bytes preproc + $(64t + |m| + 32)t$ bytes
  - Latencies: 1 preproc + 2