Seminar in Distributed Computing

Distributed Oblivious RAM for Secure Two-Party Computation

Steve Lu & Rafail Ostrovsky

Philipp Gamper

Yao's millionaires problem

- Two millionaires[1] wish to know who is richer

[1] in the following, Alice and Bob

Yao's millionaires problem

- Two millionaires[1] wish to know who is richer
- They do not want share any information about each others wealth

[1] in the following, Alice and Bob

Yao's millionaires problem

- Two millionaires[1] wish to know who is richer
- They do not want share any information about each others wealth
- How can they carry out such a conversation?

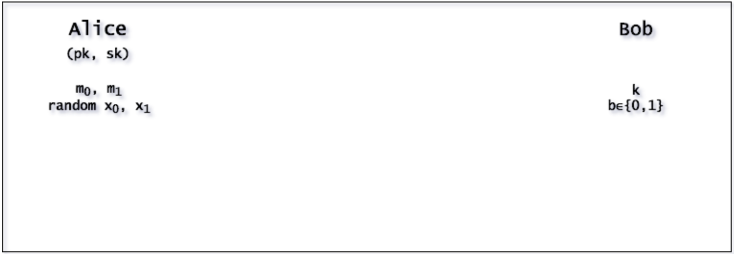
[1] in the following, Alice and Bob

Oblivious transfer (OT)

- Alice has two messages m_0 and m_1, Bob has a bit b
- Bob wishes to receive m_b, without Alice learning b
- Alice wants Bob receiving only either of the two messages

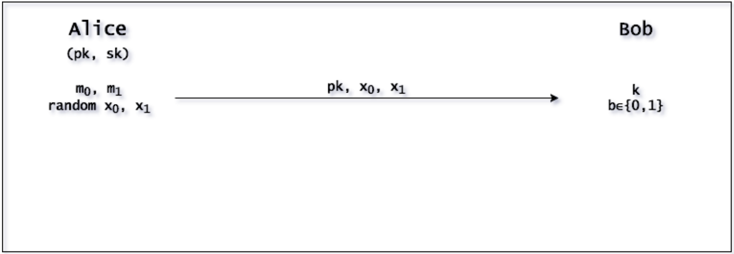
Oblivious transfer (OT)

- Alice has two messages m_0 and m_1, Bob has a bit b
- Bob wishes to receive m_b, without Alice learning b
- Alice wants Bob receiving only either of the two messages



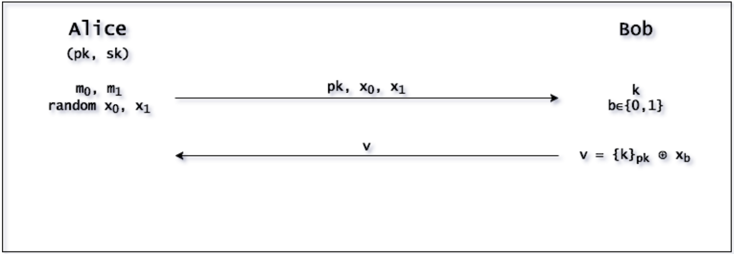
Oblivious transfer (OT)

- Alice has two messages m_0 and m_1, Bob has a bit b
- Bob wishes to receive m_b, without Alice learning b
- Alice wants Bob receiving only either of the two messages



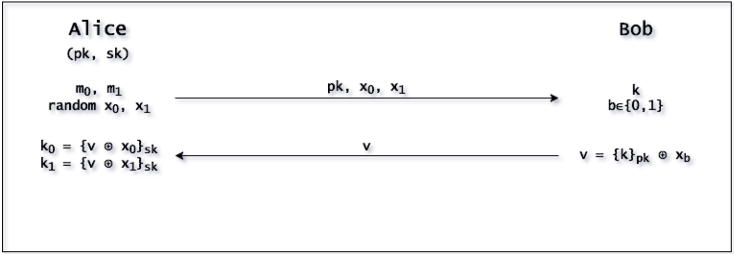
Oblivious transfer (OT)

- Alice has two messages m_0 and m_1, Bob has a bit b
- Bob wishes to receive m_b, without Alice learning b
- Alice wants Bob receiving only either of the two messages



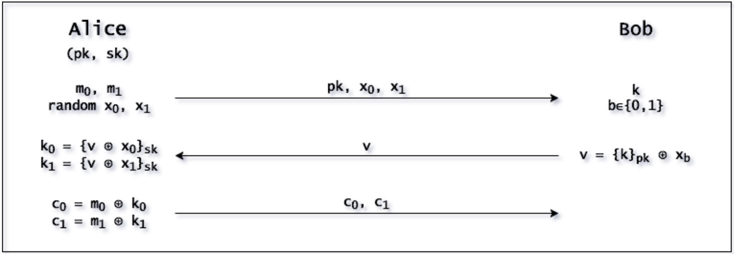
Oblivious transfer (OT)

- Alice has two messages m_0 and m_1, Bob has a bit b
- Bob wishes to receive m_b, without Alice learning b
- Alice wants Bob receiving only either of the two messages



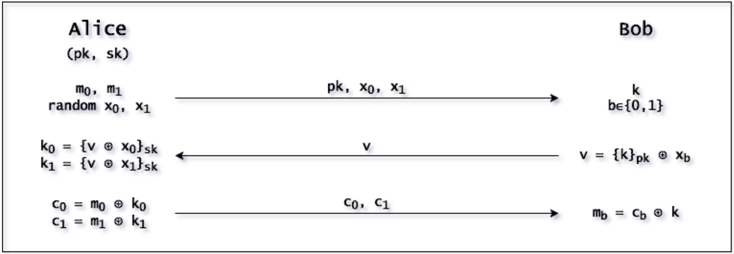
Oblivious transfer (OT)

- Alice has two messages m_0 and m_1, Bob has a bit b
- Bob wishes to receive m_b, without Alice learning b
- Alice wants Bob receiving only either of the two messages



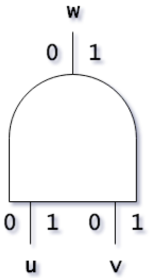
Oblivious transfer (OT)

- Alice has two messages m_0 and m_1, Bob has a bit b
- Bob wishes to receive m_b, without Alice learning b
- Alice wants Bob receiving only either of the two messages



Boolean circuits

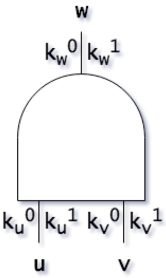
- AND-gate and its corresponding truth table



u	v	w
0	0	0
0	1	0
1	0	0
1	1	1

Garbled circuits

- garble input / output wires by assigning keys / labels to them

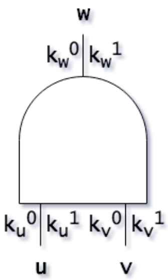


u	v	w
k_u^0	k_v^0	k_w^0
k_u^0	k_v^1	k_w^0
k_u^1	k_v^0	k_w^0
k_u^1	k_v^1	k_w^1

[2] $Enc_{a,b}(x) = Enc_a(Enc_b(x))$

Garbled circuits

- garble input / output wires by assigning keys / labels to them
- encrypt[2] output wire using the keys of the input wires



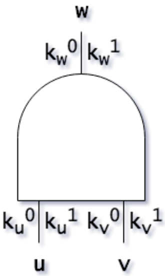
u	v	w
k_u^0	k_v^0	k_w^0
k_u^0	k_v^1	k_w^0
k_u^1	k_v^0	k_w^0
k_u^1	k_v^1	k_w^1

u	v	w
k_u^0	k_v^0	$Enc_{k_u^0,k_v^0}(k_w^0)$
k_u^0	k_v^1	$Enc_{k_u^0,k_v^1}(k_w^0)$
k_u^1	k_v^0	$Enc_{k_u^1,k_v^0}(k_w^0)$
k_u^1	k_v^1	$Enc_{k_u^1,k_v^1}(k_w^1)$

[2] $Enc_{a,b}(x) = Enc_a(Enc_b(x))$

Garbled circuits

- garble input / output wires by assigning keys / labels to them
- encrypt[2] output wire using the keys of the input wires
- randomly permute the resulting truth table



u	v	w
k_u^0	k_v^0	k_w^0
k_u^0	k_v^1	k_w^0
k_u^1	k_v^0	k_w^0
k_u^1	k_v^1	k_w^1

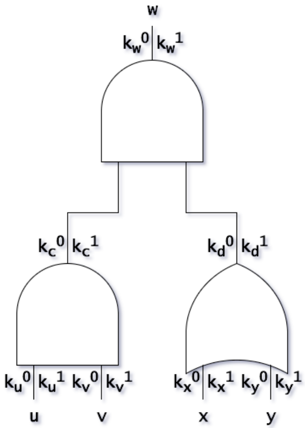
u	v	w
k_u^0	k_v^1	$Enc_{k_u^0,k_v^1}(k_w^0)$
k_u^1	k_v^1	$Enc_{k_u^1,k_v^1}(k_w^1)$
k_u^1	k_v^0	$Enc_{k_u^1,k_v^0}(k_w^0)$
k_u^0	k_v^0	$Enc_{k_u^0,k_v^0}(k_w^0)$

[2] $Enc_{a,b}(x) = Enc_a(Enc_b(x))$

Garbled circuits

- evaluate the circuit gate by gate to obtain the encrypted output

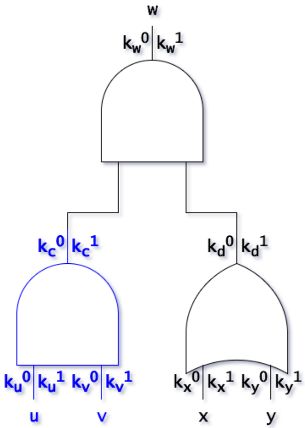
c	d	w
$Enc_{k_u^0,k_v^0}(k_c^0)$	$Enc_{k_x^0,k_y^0}(k_d^0)$	$Enc_{k_c^0,k_d^0}(k_w^0)$
$Enc_{k_u^0,k_v^1}(k_c^0)$	$Enc_{k_x^0,k_y^1}(k_d^0)$	$Enc_{k_c^0,k_d^1}(k_w^0)$
$Enc_{k_u^1,k_v^0}(k_c^0)$	$Enc_{k_x^1,k_y^0}(k_d^0)$	$Enc_{k_c^1,k_d^0}(k_w^0)$
$Enc_{k_u^1,k_v^1}(k_c^1)$	$Enc_{k_x^1,k_y^1}(k_d^1)$	$Enc_{k_c^1,k_d^1}(k_w^1)$



Garbled circuits

- evaluate the circuit gate by gate to obtain the encrypted output

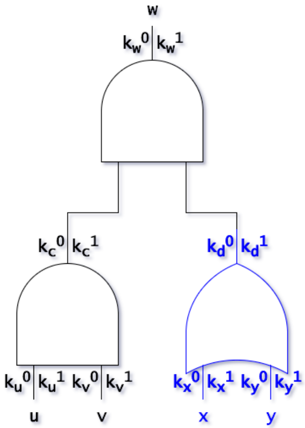
c	d	w
$Enc_{k_u^0,k_v^0}(k_c^0)$	$Enc_{k_x^0,k_y^0}(k_d^0)$	$Enc_{k_c^0,k_d^0}(k_w^0)$
$Enc_{k_u^0,k_v^1}(k_c^0)$	$Enc_{k_x^0,k_y^1}(k_d^0)$	$Enc_{k_c^0,k_d^1}(k_w^0)$
$Enc_{k_u^1,k_v^0}(k_c^0)$	$Enc_{k_x^1,k_y^0}(k_d^0)$	$Enc_{k_c^1,k_d^0}(k_w^0)$
$Enc_{k_u^1,k_v^1}(k_c^1)$	$Enc_{k_x^1,k_y^1}(k_d^1)$	$Enc_{k_c^1,k_d^1}(k_w^1)$



Garbled circuits

- evaluate the circuit gate by gate to obtain the encrypted output

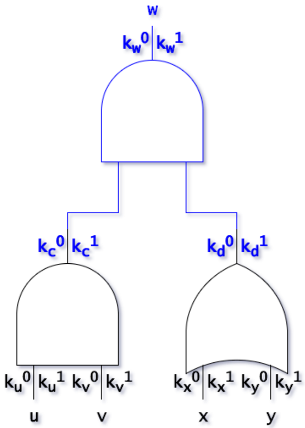
c	d	w
$Enc_{k_u^0,k_v^0}(k_c^0)$	$Enc_{k_x^0,k_y^0}(k_d^0)$	$Enc_{k_c^0,k_d^0}(k_w^0)$
$Enc_{k_u^0,k_v^1}(k_c^0)$	$Enc_{k_x^0,k_y^1}(k_d^0)$	$Enc_{k_c^0,k_d^1}(k_w^0)$
$Enc_{k_u^1,k_v^0}(k_c^0)$	$Enc_{k_x^1,k_y^0}(k_d^0)$	$Enc_{k_c^1,k_d^0}(k_w^0)$
$Enc_{k_u^1,k_v^1}(k_c^1)$	$Enc_{k_x^1,k_y^1}(k_d^1)$	$Enc_{k_c^1,k_d^1}(k_w^1)$



Garbled circuits

- evaluate the circuit gate by gate to obtain the encrypted output

c	d	w
$Enc_{k_u^0,k_v^0}(k_c^0)$	$Enc_{k_x^0,k_y^0}(k_d^0)$	$Enc_{k_c^0,k_d^0}(k_w^0)$
$Enc_{k_u^0,k_v^1}(k_c^0)$	$Enc_{k_x^0,k_y^1}(k_d^0)$	$Enc_{k_c^0,k_d^1}(k_w^0)$
$Enc_{k_u^1,k_v^0}(k_c^0)$	$Enc_{k_x^1,k_y^0}(k_d^0)$	$Enc_{k_c^1,k_d^0}(k_w^0)$
$Enc_{k_u^1,k_v^1}(k_c^1)$	$Enc_{k_x^1,k_y^1}(k_d^1)$	$Enc_{k_c^1,k_d^1}(k_w^1)$

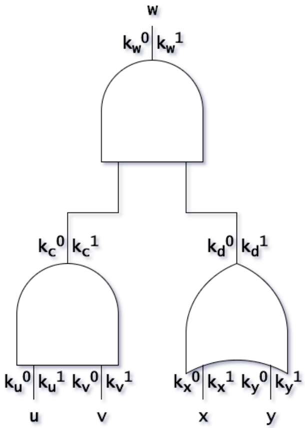


Garbled circuits

- evaluate the circuit gate by gate to obtain the encrypted output

c	d	w
$Enc_{k_u^0,k_v^0}(k_c^0)$	$Enc_{k_x^0,k_y^0}(k_d^0)$	$Enc_{k_c^0,k_d^0}(k_w^0)$
$Enc_{k_u^0,k_v^1}(k_c^0)$	$Enc_{k_x^0,k_y^1}(k_d^0)$	$Enc_{k_c^0,k_d^1}(k_w^0)$
$Enc_{k_u^1,k_v^0}(k_c^0)$	$Enc_{k_x^1,k_y^0}(k_d^0)$	$Enc_{k_c^1,k_d^0}(k_w^0)$
$Enc_{k_u^1,k_v^1}(k_c^1)$	$Enc_{k_x^1,k_y^1}(k_d^1)$	$Enc_{k_c^1,k_d^1}(k_w^1)$

- Output translation $[(0, k_w^0), (1, k_w^1)]$



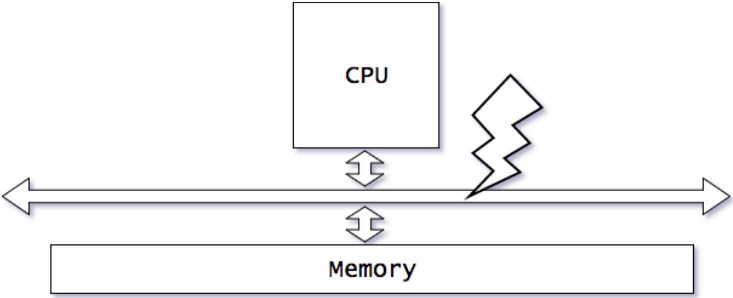
A solution to the millionaires problem

1. Alice generates a garbled full adder circuit that outputs the carry flag
2. Alice sends the circuit to Bob along with her encrypted input
3. Bob receives his encrypted inputs using oblivious transfer
4. Bob evaluates the circuit gate by gate to obtain his output
5. Alice and Bob communicate to learn the output

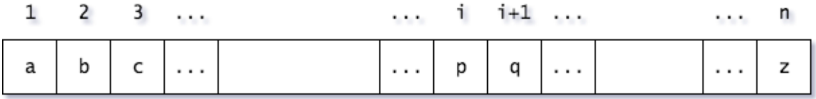
Oblivious RAM (ORAM)

Oblivious RAM (ORAM)

Def (informal): *The sequence of memory access of an oblivious RAM reveals no information about the input, beyond the running time for the input*

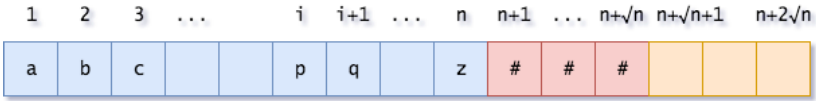


Trivial ORAM

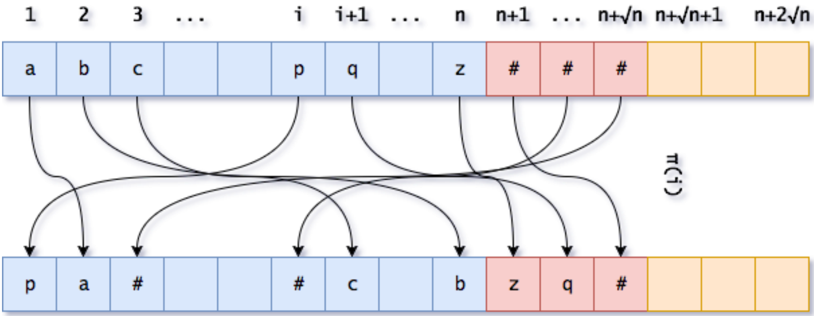


- for every operation scan through entire memory
- obviously hides the *access pattern*
- BUT causes $\mathcal{O}(n)$ overhead

The „square root" solution

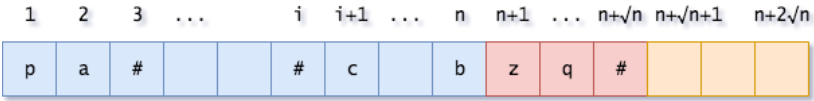


The „square root" solution - initialization

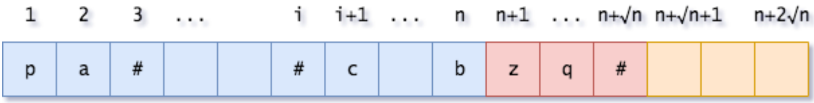


- randomly permute memory cells 1 to $n + \sqrt{n}$ using a PRF $\pi(i)$

The „square root" solution - *i*-th step (i = 1, v = 3)

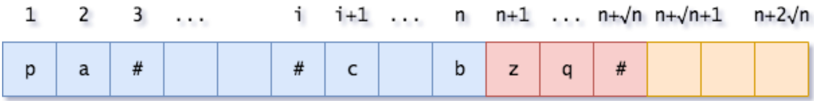


The „square root" solution - *i*-th step ($i = 1, v = 3$)



1. simulate $\sqrt{n}$ memory accesses by reading cells $n + \sqrt{n} + 1$ to $n + 2\sqrt{n}$

The „square root” solution - i-th step ($i = 1$, $v = 3$)



1. simulate $\sqrt{n}$ memory accesses by reading cells $n + \sqrt{n} + 1$ to $n + 2\sqrt{n}$
2. if v-th cell was found, access next dummy cell $\pi(n + i)$ else retrieve it from $\pi(v)$

The „square root” solution - i-th step ($i = 1, v = 3$)

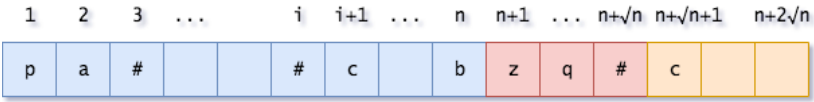
1	2	3	...		i	i+1	...	n	n+1	...	n+√n	n+√n+1		n+2√n
p	a	#			#	c		b	z	q	#	c		

1. simulate $\sqrt{n}$ memory accesses by reading cells $n + \sqrt{n} + 1$ to $n + 2\sqrt{n}$
2. if v-th cell was found, access next dummy cell $\pi(n + i)$ else retrieve it from $\pi(v)$
3. keep the value of the v-th cell in the $n + \sqrt{n} + i$-th cell

The „square root” solution - *i*-th step (i = 2, v = n)

1	2	3	...		i	i+1	...	n	n+1	...	$n+\sqrt{n}$	$n+\sqrt{n}+1$		$n+2\sqrt{n}$
p	a	#			#	c		b	z	q	#	c		

The „square root” solution - *i*-th step ($i = 2$, $v = n$)



- first access to cell v, retrieve from $\pi(v)$ after simulating $\sqrt{n}$ memory accesses

The „square root” solution - i-th step ($i = 2$, $v = n$)

1	2	3	...		i	i+1	...	n	n+1	...	n+√n	n+√n+1		n+2√n
p	a	#			#	c		b	z	q	#	c	z	

- first access to cell v, retrieve from $\pi(v)$ after simulating $\sqrt{n}$ memory accesses
- keep the value of the v-th cell in the $n + \sqrt{n} + i$-th cell

The „square root” solution - *i*-th step (i = 3, v = 3)

1	2	3	...		i	$i+1$	...	n	$n+1$	...	$n+\sqrt{n}$	$n+\sqrt{n}+1$		$n+2\sqrt{n}$
p	a	#			#	c		b	z	q	#	c	z	

The „square root” solution - *i*-th step ($i = 3$, $v = 3$)

1	2	3	...		i	i+1	...	n	n+1	...	n+√n	n+√n+1		n+2√n
p	a	#			#	c		b	z	q	#	c	z	

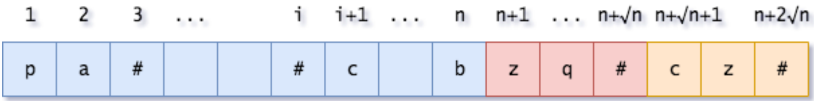
- the v-th cell has been read before, access dummy cell $\pi(n+i)$

The „square root” solution - i-th step ($i = 3$, $v = 3$)

1	2	3	...		i	i+1	...	n	n+1	...	n+√n	n+√n+1		n+2√n
p	a	#			#	c		b	z	q	#	c	z	#

- the v-th cell has been read before, access dummy cell $\pi(n+i)$
- after $\sqrt{n}$ queries the cache is full

The „square root” solution - i-th step ($i = 3$, $v = 3$)



- the v-th cell has been read before, access dummy cell $\pi(n+i)$
- after $\sqrt{n}$ queries the cache is full
- reshuffling called *oblivious sorting* required

Oblivious sorting

Def (informal): *A sorting algorithm is called oblivious iff the sequence of compare operations is independent of the input.*

The general, very informal idea of oblivious sorting is as follows

- assign random tags to each memory cell
- sort the cells according to the tag

e.g., *Bubble Sort* is oblivious, while *Quick Sort* is not

The „square root" solution - Analysis

Overhead to perform n queries

- each query requires $\sqrt{n}$ memory accesses
- there are $\sqrt{n}$ queries per round, what requires $\sqrt{n} \times \sqrt{n} = n$ memory accesses
- $\sqrt{n}$-rounds $\times \sqrt{n}$-queries results in $\sum_{i=1}^{\sqrt{n}} n = n\sqrt{n}$, thus $\mathcal{O}(n\sqrt{n})$
- oblivious sorting requires $\mathcal{O}(nlog(n))$, and $\mathcal{O}(\sqrt{n}log(n)n)$ in total
- the overall overhead is $\mathcal{O}(\sqrt{n}log(n))$

The „square root" solution - Analysis

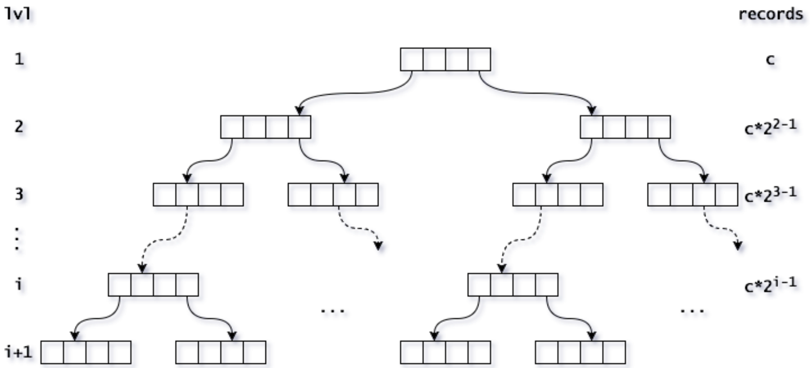
Overhead to perform n queries

- each query requires $\sqrt{n}$ memory accesses
- there are $\sqrt{n}$ queries per round, what requires $\sqrt{n} \times \sqrt{n} = n$ memory accesses
- $\sqrt{n}$-rounds $\times \sqrt{n}$-queries results in $\sum_{i=1}^{\sqrt{n}} n = n\sqrt{n}$, thus $\mathcal{O}(n\sqrt{n})$
- oblivious sorting requires $\mathcal{O}(nlog(n))$, and $\mathcal{O}(\sqrt{n}log(n)n)$ in total
- the overall overhead is $\mathcal{O}(\sqrt{n}log(n))$

Hence the ORAM simulation is dominated by the oblivious sorting

Secure two party computation using ORAM

ORAM using a hierarchical data structure



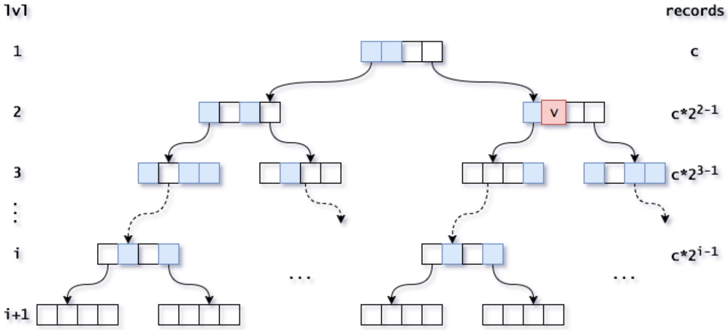
ORAM using a hierarchical data structure

the idea remains the same as in the „square root” solution

1. fetch some records in order to obtain (v, x)
2. check whether x has been found or not
3. retrieve directly from $\pi(v)$ or do dummy access respectively
4. re-encrypt x and store back
5. reshuffle if „cache”, i.e., the root bucket, becomes full

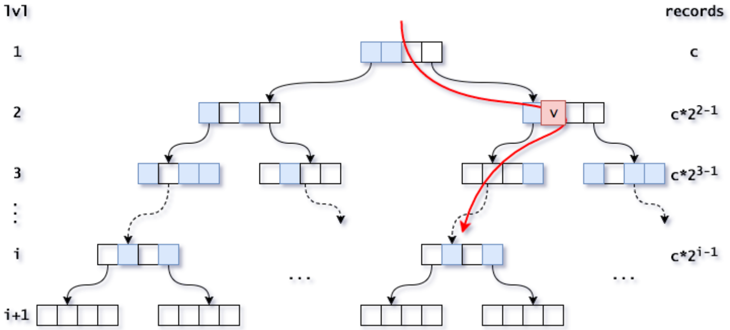
The query (simplified)

- first query to cell *v*



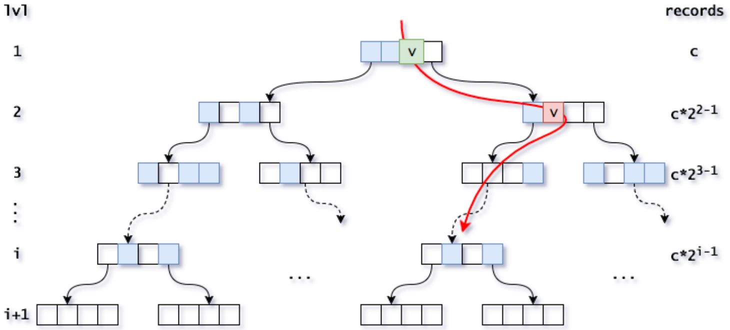
The query (simplified)

- always traverse tree until reaching a leaf



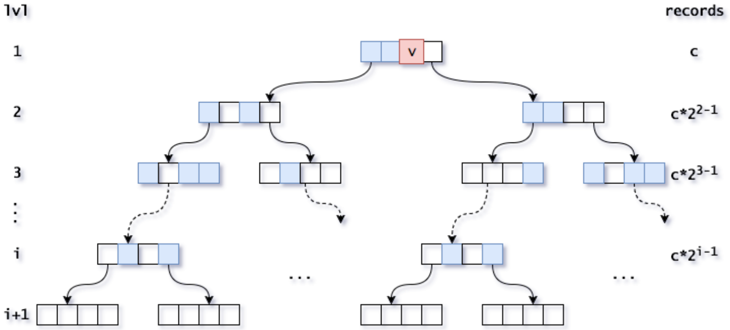
The query (simplified)

- re-insert cell *v* in the next empty cell in the root bucket



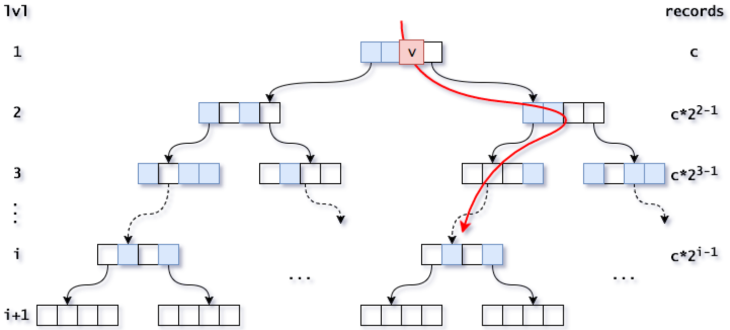
The query (simplified)

- further queries to cell *v*



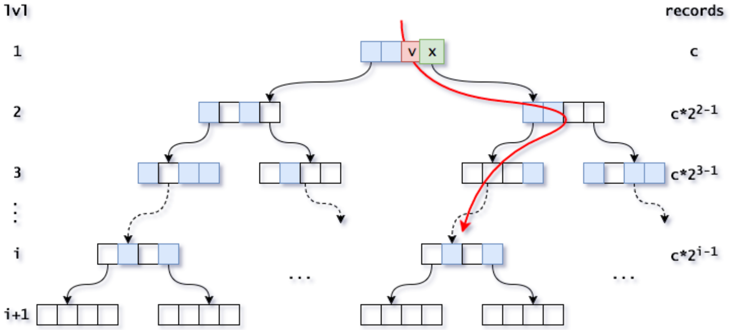
The query (simplified)

- again fetch until reaching a leaf node



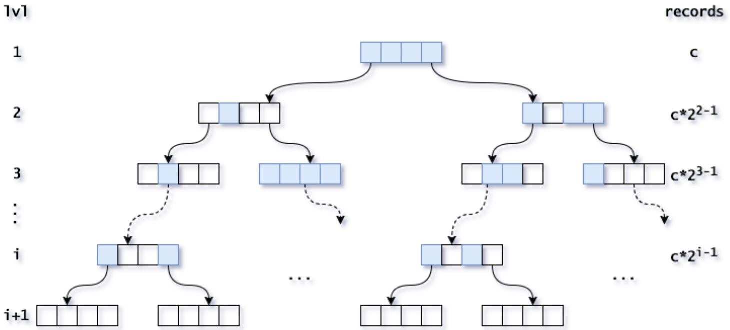
The query (simplified)

- insert dummy cell, as *v* is already in the root bucket



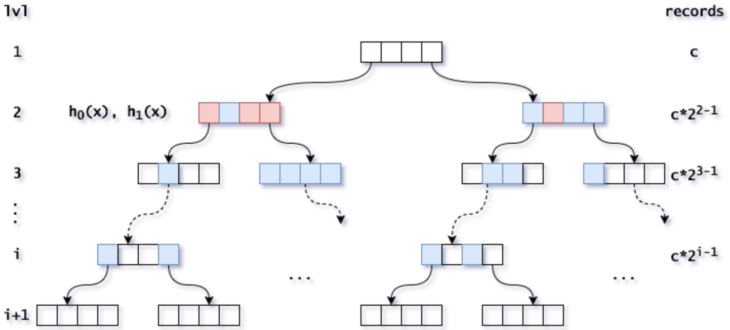
The reshuffling (simplified)

- reshuffling required, if the root bucket becomes full



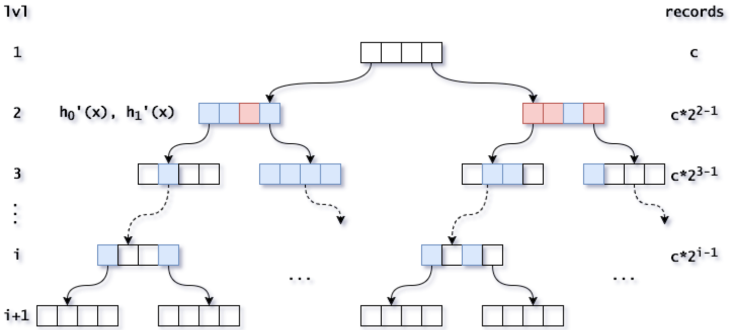
The reshuffling (simplified)

- all cells of the root bucket are pushed down to the 2nd-level



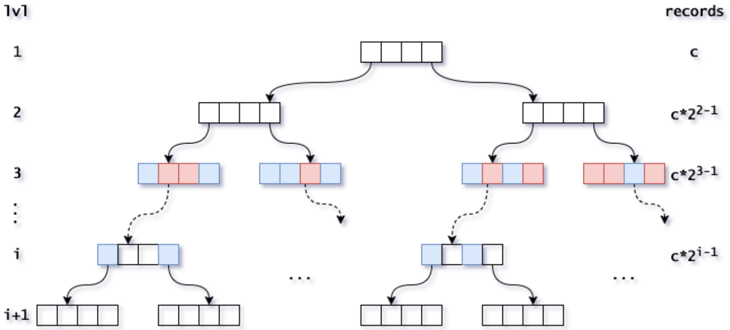
The reshuffling (simplified)

- while reshuffling level *j*, two new hash functions are chosen



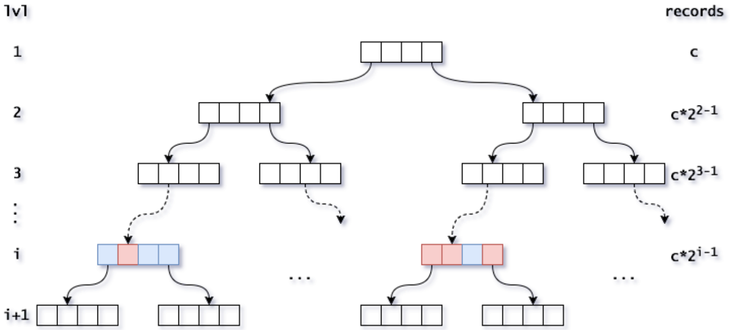
The reshuffling (simplified)

- if $j + 1$-th level becomes full, it is reshuffled as well



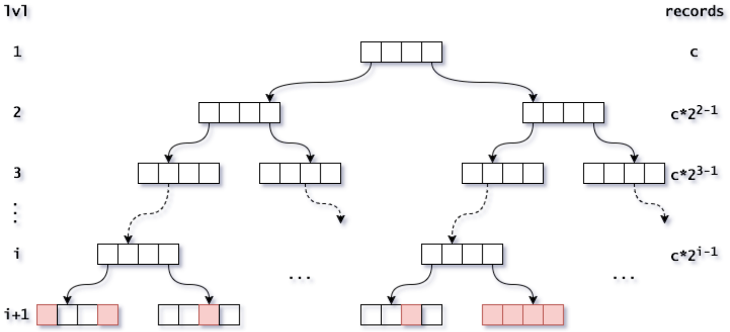
The reshuffling (simplified)

- if the 2nd to *i*-th level are half full, level *i* eventually becomes full



The reshuffling (simplified)

- hence, the $i+1$-th level ends up half full



Key ingredients

- levels are alternating distributed on the two server
- avoid oblivious sorting
- use „tagging", that is $PRF(i, e_i, v)$, where e is the *epoch*, i the *level* and v the index of the record
- Cuckoo hashing with a stash to cause the buckets overflowing with negligible probability

Analysis

- $\mathcal{O}(log(n))$ computational overhead, if using a buckets of size $3 * log(n)/log(log(n))$
- $\mathcal{O}(1)$ client storage
- two servers using $\mathcal{O}(n)$ storage each
- negligible probability of an attacker is able to distinguish between two query sequences

Secure two party computation

Two parties wish to compute some function $f(x, y)$ on their inputs x and y

Secure two party computation

Two parties wish to compute some function $f(x, y)$ on their inputs x and y

- let both parties play the role of one server each
- the client is *shared* between the two parties using *secret sharing*
- only build atomic opration on the ORAM in circuits
- simulate the underlying circuit of f using ORAM
- communicate to learn the output

Conclusion

Conclusion

- we have seen a multi-server model for oblivious RAM using $\mathcal{O}(1)$ client and $\mathcal{O}(n)$ server storage resulting in a only $\mathcal{O}(log(n))$ computational overhead
- a two-party secure RAM computation protocol, that is more efficient than existing construction

References

- Towards a theory of software protection and simulation by oblivious RAMs – *Oded Goldreich*
- Protocols for secure computations – *Andrew Chi-Chih Yao*
- Software protection and simulation on oblivious RAMs – *Oded Goldreich, Rafail Ostrovsky*
- Efficient computation on oblivious rams – *Rafail Ostrovsky*
- Private information storage – *Rafail Ostrovsky, VictorShoup*
- More robust hashing: Cuckoo hashing with a stash – *Adam Kirsch, Michael Mitzenmacher, and Udi Wieder*