CS 798
Privacy in Computation and Communication

Module 4
Privacy in Computation: Trusted Hardware

Spring 2024

Trusted hardware

Recall the three main ways to achieve privacy in computation:

- Distributed trust
- Trusted hardware
- Homomorphic encryption

Trusted hardware

Recall the three main ways to achieve privacy in computation:

- Distributed trust
- Trusted hardware
- Homomorphic encryption

Motivation

- Maybe you would be OK with sending your private data to a central computer for processing if. . .
  - you could be assured exactly what processing will be done
  - your data could not be stored or processed except for the above
  - no person (or machine) would be able to see your data or any derivative data except for the above
- . . . maybe?

Trusted hardware

- Have a trusted part of a computer that can execute (possibly a small bit of) code in a way that can't be interfered with by the rest of the computer
    - "Trusted Execution Environment" (TEE)
    - Remember what "trusted" means in security!
- Technologies like these date back over 25 years
- Many hardware manufacturers have a version of this
    - Intel SGX, ARM TrustZone, AMD SEV, etc.

Threat model

- The details vary from system to system, but typically:
  - A user should be able to communicate with a program running on a remote computer. . .
  - with assurance as to what program is running. . .
  - and that even someone controlling the remote operating system cannot see the data the program is receiving, processing, or sending
- Outside the threat model:
  - Opening up the physical CPU package (though some systems may do their best to destroy any secrets if physical tampering is detected)
  - Availability, metadata

Intel SGX

- We'll use Intel SGX as a running example of a trusted hardware system, but others are similar
  - Our example will use Linux as the OS
- Supported by server-class (Xeon) processors, but not (most) client-class Intel processors
- There are a couple of variants (e.g., SGX1, SGX2); we won't worry about the distinctions
- *Note:* the client does *not* need a CPU with SGX support!

Enclaves

- On the server side, SGX-enabled programs have two parts:

- The untrusted application code
    - This is just a normal Linux program: it can take input from the command line, terminal, Internet sockets, whatever
    - In the threat model, this code is assumed to be malicious

- The trusted code running in an *enclave*
    - The untrusted code can interact with the enclave through APIs the enclave defines, but *cannot* see the data “inside” the enclave
    - The enclave *cannot* perform any system calls at all!

Enclaves

- Enclaves have memory assigned to them
- This memory is *encrypted* by the CPU
    - Remember how AES is implemented in hardware in modern CPUs?
- Not even a malicious operating system can read the contents of this memory
- But it can see what memory *addresses* are being accessed

  $\Rightarrow$ The metadata

Attestation

- Enclaves can create public/private key pairs
- There is a procedure called *remote attestation* that allows a client to verify that a particular public key was created by an enclave on a remote machine running a particular piece of trusted code
- Then the client can use that public key to communicate with the enclave with confidentiality, integrity, and authenticity
    - Even though the communication goes *through* the untrusted application

Attacks on SGX

- There have been a number of attacks on the security of SGX in the past
- Some (e.g., SgxPectre, Foreshadow) violate the SGX (and indeed the CPU) threat model, and Intel issues patches to fix them
- Others (e.g., MemBuster, TeeJam) do not violate the SGX threat model, and enclave authors are on their own to mitigate them
    - These typically use side channels to reveal to the (malicious) operating system things like memory addresses being accessed, or the sequence of instructions being executed

Mitigating side channels

- Some data is *public*
  - e.g., the *size* of the input
- Some data is *private*
  - e.g., data collected from users
- When you write a program to be run in an enclave, you need to ensure that there are no side channels that can leak private data

Mitigating side channels

- Some data is *public*
    - e.g., the *size* of the input
- Some data is *private*
    - e.g., data collected from users
- When you write a program to be run in an enclave, you need to ensure that there are no side channels that can leak private data

Mitigating side channels

- The big ones:
  - Instruction sequences
  - Addresses of memory accesses
- These must *not* depend on any private data!
- So loops where the loop bounds are the size of the input are fine
- But loops where the loop bounds depend on the user data are not

Oblivious algorithms

- In the trusted hardware setting, a program that, for a given set of public data (e.g., the size of the input), always:
  - executes the same sequence of instructions, and
  - accesses the same memory addresses for every read and write

  is a (fully) oblivious algorithm

- Note that the *values* written to or read from memory *are* allowed to depend on the private data
  - The memory encryption will protect those

Oblivious algorithms

- How do oblivious algorithms in the trusted hardware setting compare to those in the distributed trust setting?

- In the distributed trust setting, the parties *have no knowledge* of the private data, so they *cannot* behave in a way that accidentally reveals it

- In the trusted hardware setting, the enclave *does* see the private data, so it's actually pretty easy to slip up and accidentally create a side channel that reveals it
    - The programmer has to be much more careful
    - The *compiler* may even slip you up, even if you were careful!

Oblivious conditionals

- If you have an if/then/else with the branch test involving private data, how do you make that oblivious?
- The basic idea is the same as in the distributed trust setting:
  - Execute both branches
  - Use an oblivious multiplexer to choose which result to keep and which to throw away
- You'll of course want to write your code to minimize the number of such conditional branches (with private conditions) in the first place!

Oblivious multiplexers

- How do you implement the oblivious multiplexer?
  - You might think just, e.g., `y = b ? x1 : x0;`
- The above is *not* guaranteed to be oblivious!
  - The compiler may decide to put a conditional branch in there, or it may not, or it may depend on the optimization level, or the compiler version, or any number of things
- You may need to resort to inline assembly, particularly the `cmov` instruction family

Oblivious multiplexers

- For example:

```
    y = b ? x1 : x0;
```

  becomes

```
    mov %[x0], %[y]
    test %[b], %[b]
    cmovnz %[x1], %[y]
```

- This version *is* oblivious

Oblivious multiplexers

- For example:

  ```
  y = b ? x1 : x0;
  ```

  becomes

`mov %[x0], %[y]`	$y \leftarrow x0$
`test %[b], %[b]`	Test if b is 0
`cmovnz %[x1], %[y]`	$y \leftarrow x1$ *if* $b \neq 0$

- This version *is* oblivious

Oblivious memory manipulation

- The previous slide looked at data in registers. What about memory?
- There are two cases:
    - You want to maybe (depending on private data) update a variable in memory at a public address
    - You want to maybe (depending on private data) update a variable in memory at an address that itself depends on private data
- We'll look at the first (easier) case now, and the second (much harder) case later in the module

Oblivious swaps

- Example: there are two blocks of data in memory at *public* addresses (and sizes)
- The *contents* of the blocks are private
- You have a *private* control bit, and you want to *swap* the contents of the blocks of memory if the bit is 1, and not change memory if the bit is 0
    - But you have to do this obliviously, so that the adversarial OS can't even tell whether the control bit was 1 or 0
- This is an *oblivious swap*, which is a key component of many oblivious data manipulation algorithms, such as oblivious sorting (coming soon in this module)

Oblivious swaps

- The first thing to notice is that if you're trying to obliviously swap x and y, each of say 200 bytes, based on the control bit b, then you can do it in chunks:

```
oswap(x, y, b, 200):
   oswap_8bytes(x, y, b)
   oswap_8bytes(x+8, y+8, b)
   oswap_8bytes(x+16, y+16, b)
   ...
   oswap_8bytes(x+192, y+192, b)
```

Oblivious swaps

- The first thing to notice is that if you're trying to obliviously swap x and y, each of say 200 bytes, based on the control bit b, then you can do it in chunks:

```
oswap(x, y, b, 200):
    for (i=0; i<200; i+=8):
        oswap_8bytes(x+i, y+i, b)
```

Oblivious swaps

- Then for each 8 bytes, the strategy is:
  - Read both 8-byte memory locations into registers
  - Use `cmov` to obliviously swap the registers, depending on the control bit b
  - Write both registers back to memory

```
test %[b], %[b]
mov (%[y]), %r10
mov (%[y]), %r11
mov (%[x]), %r12
cmovnz %r12, %r10
cmovnz %r11, %r12
mov %r10, (%[y])
mov %r12, (%[x])
```

Oblivious swaps

- Ideally, you'd have a library that implements this for you, of course
- Then you just have to call the appropriate oblivious functions when you want to do memory manipulations depending on private data
- Note the performance cost (will be important later): each 8 bytes executes a constant number of instructions, so to obliviously swap two buffers of size β bytes takes $\mathcal{O}(\beta)$ time

Oblivious data analysis

- Consider the example of *privacy preserving telemetry*: a hardware or software maker (say Google) wants to collect statistical information about how often users use certain features, how often crashes happen, what websites are commonly visited, etc.
- Importantly, this statistical information is *independent* of the identities of the users
    - It doesn't matter *which* user's browser crashed; you're just counting *how many* crashes happened, for example

Oblivious data analysis

- When collecting this data, an important first step is to break the connection between the data and the user that submitted it
    - You want to do this in a way that the users can be *assured* that their usage patterns aren't linked to their identities / accounts

- Strategy: have users submit their data encrypted into a TEE, which *shuffles* the submissions in a large batch so that no one can tell which user submitted which data once it comes out of the TEE

- Google's Prochlo system does this, for example

Oblivious shuffles and sorts

- But remember that to be secure against a potentially malicious operating system, this shuffling must be done *obliviously*
    - An adversary who compromises the OS, watching what memory addresses are being read and written to, should *not* be able to tell which outputs of the shuffle corresponded to which inputs
- So we need to use algorithms for *oblivious shuffles*
- Similarly, sometimes we need to be able to do *oblivious sorts*, so that we do not reveal which input item ended up at each output location after the sort
    - Applications to private database joins, privacy preserving federated learning, and more

Oblivious shuffles and sorts

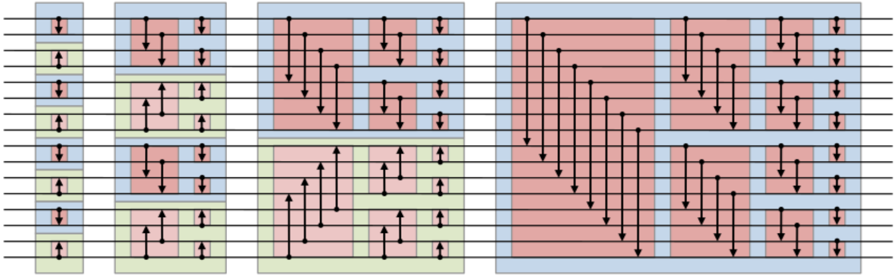
- Oblivious shuffles and sorts are closely related
- If you have an algorithm for oblivious sorts, you can make an oblivious shuffle with just a little extra work:
    - For each item in the array you want to shuffle, prepend a random sorting key
    - Obliviously sort the items on those sorting keys
    - Throw away the sorting keys

Oblivious shuffles and sorts

- If you have an algorithm for oblivious shuffles, you can make an oblivious sort with some extra work:
    - Obliviously shuffle the input data; then use any *non-oblivious* sorting algorithm (quicksort, heapsort, whatever) to sort the result
    - The oblivious shuffle breaks the link between the input and output data items, so even if the adversary can tell which inputs to the quicksort (for example) ended up at which output, they learn nothing about which original inputs (before the shuffle) they were
    - This is called the “Scramble-then-Compute” paradigm

Sorting networks

- A *sorting network* is a way to do oblivious sorting

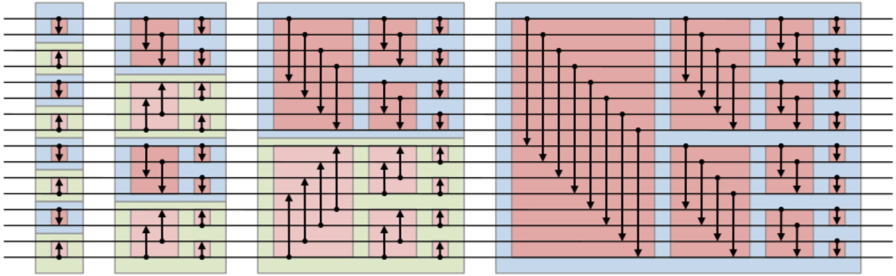


https://en.wikipedia.org/wiki/Bitonic_sorter

- Each arrow is an *oblivious swap*, with the control bit set so that the larger element ends up at the head of the arrow

Sorting networks

- A *sorting network* is a way to do oblivious sorting

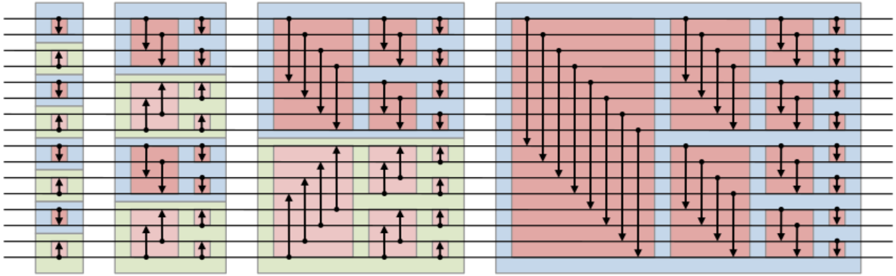


https://en.wikipedia.org/wiki/Bitonic_sorter

- Each oblivious swap operates on a predetermined pair of the inputs, independent of the data

Sorting networks

- A *sorting network* is a way to do oblivious sorting

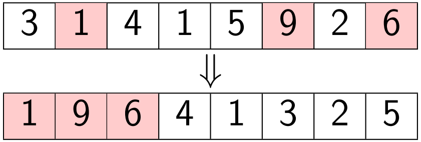


https://en.wikipedia.org/wiki/Bitonic_sorter

- Cost to bitonic sort n items of size β:
  $\frac{n}{4}(\lg^2 n + \lg n)$ oblivious swaps of size β

Oblivious compaction

- Aside: another useful subroutine for oblivious data processing is *oblivious compaction*
- For processing a subset of data without revealing what subset
- You start with an array of data items, some of which are “marked”
- The operation is to move the marked items to the beginning of the array (preserving their order)
    - The unmarked items go after, in any order

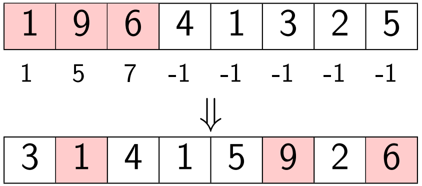


Oblivious compaction

- Cost:
- Traditionally use Goodrich's algorithm (2011): around $n(\lg n - 2)$ oblivious swaps of size β
- Newer: Sasy et al.'s Oblivious Recursive Compaction (ORCompact, 2022): $\frac{n}{2} \lg n$ oblivious swaps of size β

Oblivious expansion

- If you run an oblivious compaction algorithm “backwards” (perform the oblivious swaps in the reverse order), you get an *oblivious expansion* algorithm:



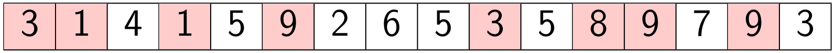
- Cost: same as oblivious compaction

ORShuffle

3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

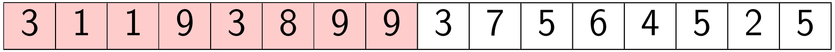
ORShuffle

MarkHalf



ORShuffle

MarkHalf + ORCompact



ORShuffle

MarkHalf + ORCompact + Recurse Left and Right = ORShuffle

9	9	8	3	3	1	9	1	5	5	6	4	2	5	7	3

ORShuffle

- ORShuffle cost: $\frac{n}{4}(\lg^2 n + \lg n)$ oblivious swaps of size β
- Compare Bitonic sort with random labels: $\frac{n}{4}(\lg^2 n + \lg n)$ oblivious swaps of size $\beta + \text{sizeof(label)}$
- For small items, ORShuffle is $2\times$ faster; for large items, only very slightly faster

Permutation networks

- A *permutation network* is similar to a sorting network:
  - A fixed set of oblivious swaps each operating on two predetermined items, each with a control bit
- But: a permutation network can apply *any* desired permutation to the input items, not just the particular permutation that sorts them
- A permutation network has significantly fewer oblivious swaps than a sorting network

Permutation networks

- Recall that in a sorting network, the control bits are set automatically by comparing the two items in order to make the larger item end up at the "head of the arrow"
- In a permutation network, the control bits need to be set by a separate algorithm that takes as input the desired permutation

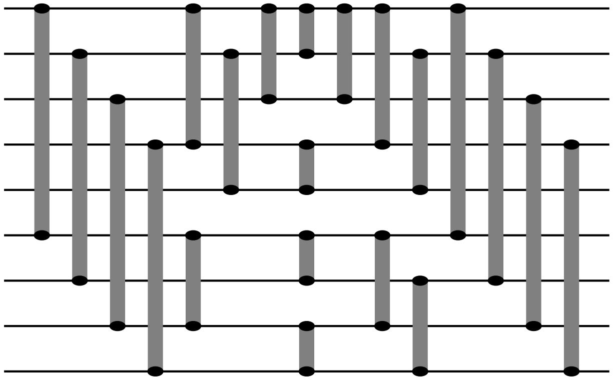
Permutation networks

Using a permutation network therefore typically has three phases:

- Determine the permutation you want
- Given that permutation, determine what the control bits should be
- Apply the permutation by executing the oblivious swaps with those control bits

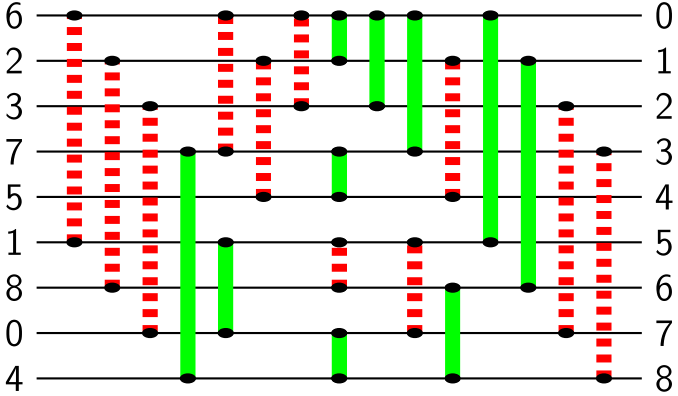
Waksman networks

- A Waksman network is the most common type of permutation network



Waksman networks

- A Waksman network is the most common type of permutation network



Waksman networks

- A Waksman network on n inputs has approximately $n \lg n - n + 1$ oblivious swaps
    - Exactly this number when n is a power of 2

- Compare bitonic sorting network: $\frac{n}{4}(\lg^2 n + \lg n)$ oblivious swaps
    - A factor of more than $\frac{1}{4} \lg n$ more than a Waksman network

Determining the permutation

- The first of the three phases is to determine what permutation you want to apply to your data items
- We'll use the example of sorting for now
- Given an array of data items, what permutation do you need to apply to put them in sorted order?
- We'll assume each data item is large (size β), but has a small sorting key
    - Transaction id, userid, something like that

Determining the permutation

271	828	182	845	904	523	536	28

- Copy the (small) sorting keys into an array

Determining the permutation

271	828	182	845	904	523	536	28
0	1	2	3	4	5	6	7

- Copy the (small) sorting keys into an array
- Attach the original positions in the array to each item

Determining the permutation

28	182	271	523	536	828	845	904
7	2	0	5	6	1	3	4

- Copy the (small) sorting keys into an array
- Attach the original positions in the array to each item
- Obliviously sort the array

Determining the permutation

7	2	0	5	6	1	3	4

- Copy the (small) sorting keys into an array
- Attach the original positions in the array to each item
- Obliviously sort the array
- The position markers that came along with the sort indicate the desired permutation
    - Original item 7 is destined for position 0
    - Original item 2 is destined for position 1
    - etc.

Determining the permutation

7	2	0	5	6	1	3	4

- Copy the (small) sorting keys into an array
- Attach the original positions in the array to each item
- Obliviously sort the array
- The position markers that came along with the sort indicate the desired permutation
  - Original item 7 is destined for position 0
  - Original item 2 is destined for position 1
  - etc.

Determining the permutation

- What if we wanted to shuffle the data items instead of sorting them?
- Just use a uniformly random permutation:
    - Initialize an array with 0, 1, 2, $\ldots n-1$
    - Obliviously shuffle the array

Setting the control bits

- There are a number of algorithms for the second phase: setting the control bits of a Waksman network to a desired permutation
- Caveat: you need to implement the control bit setting algorithm *itself* obliviously!
    - The adversary controlling the OS and watching all memory accesses must not be able to determine what the permutation is or what the resulting control bits are
- Cost: you can obliviously set the control bits of a Waksman network for a given permutation in $\mathcal{O}(n \lg^3 n)$ time

Setting the control bits

- Note: if you want to set the Waksman network to a uniform random permutation (for shuffling), you *cannot* just set each control bit uniformly at random!
- This does not yield a uniform random permutation
- There are $n!$ possible permutations, and the number of ways to set k control bits is 2^k, so it is not possible that each permutation results from the same number of settings of the control bits

Applying the permutation

- The third phase is easy: given the control bits, applying the permutation is just performing the oblivious swaps in the order specified by the Waksman network layout
- To apply a Waksman network on n items of size β, the cost is $\mathcal{O}(\beta n \lg n)$

Runtime

- Observe that the cost of phases 1 and 2 together are $\mathcal{O}(n \lg^3 n)$, while phase 3 is $\mathcal{O}(\beta n \lg n)$
- Note there is no β in the phase 1 and 2 cost!
- Also note that if you know the permutation *before* you receive the data, you can perform phases 1 and 2 as a *preprocessing* phase
    - This is always true when shuffling (a known *number* of items), for example

Runtime

Shuffling	Preproccessing	Online
Bitonic shuffle		$\mathcal{O}(\beta n \lg^2 n)$
WaksShuffle	$\mathcal{O}(n \lg^3 n)$	$\mathcal{O}(\beta n \lg n)$

Sorting	Preprocessing	Online
Bitonic sort		$\mathcal{O}(\beta n \lg^2 n)$
WaksSort		$\mathcal{O}(n \lg^3 n) + \mathcal{O}(\beta n \lg n)$
WaksShuffle + QS	$\mathcal{O}(n \lg^3 n)$	$\mathcal{O}(\beta n \lg n)$

- WaksShuffle (+ QS) always better in online time than Bitonic
- WaksSort, WaksShuffle better in total time when $\beta = \omega(\lg n)$ (concretely, $\beta > 1400\,\mathrm{B}$ for a wide range of values of n)

Accessing memory at private addresses

- Now we move on to the harder case: your algorithm (running in a TEE) needs to access memory (say an element of an array) where the *address is itself private*
  - Look up a user's account information without revealing which user it is, for example

- You should not even reveal if two accesses were to the *same* or *different* array elements / addresses

Linear scan

- If the array is small, a simple *linear scan* is likely the best way
- Read *every* element of the array one at a time (from the start to the end of the array), obliviously keeping a copy (using `cmov` for example) of only the record you're actually looking for
- For writing, read every element one at a time into a buffer, obliviously modify the buffer if it's the item you're interested in (`cmov` again), and write it back to the array
- Cost (n items of size β): $\mathcal{O}(\beta n)$ per oblivious read or write operation

Oblivious RAMs

- If n is large, linear scan will perform very poorly
- The typical solution is to use *Oblivious RAMs* (ORAMs)
- Originally designed for the client-server setting:
  - Client outsources data storage to a server
  - Client can read and write blocks of storage on the server by specifying a block *address*
  - Client wants to run an algorithm that accesses blocks as a RAM, *without* the server learning what blocks are being read or written

  $\Rightarrow$ The address of any given block must change every time the block is accessed

Oblivious RAMs

- There are n data blocks, each with an id number from 0 to $n - 1$, and each of size β
- Blocks stored on the server can be *real* blocks (one of the n data blocks) or *dummy* blocks
- Blocks stored on the server are encrypted
  - The server cannot tell the id number or contents of any given block, or even if the block is a real or dummy block

Comparing ORAMs and PIR

	ORAM	PIR
Operations	Read / write	Read only
Server computation	$\mathcal{O}(\beta \lg n)$	$\mathcal{O}(\beta n)$
Number of clients	1	unlimited

- ORAMs are limited to one client because the blocks are encrypted, and only one client can know the encryption key

Path ORAM

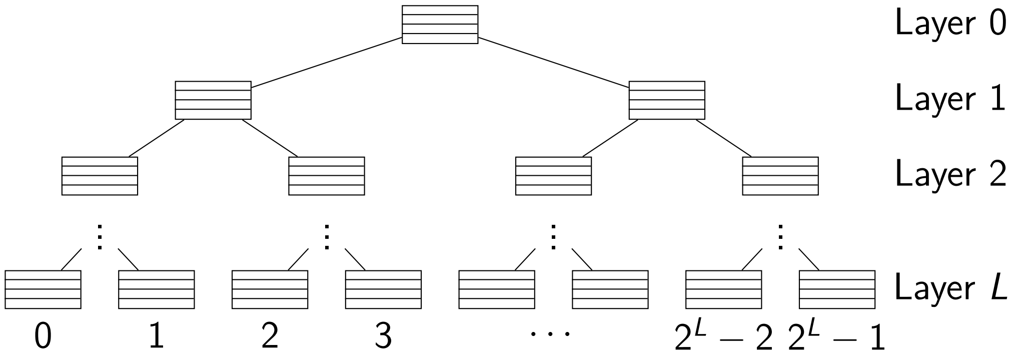
- A simple ORAM protocol is called *Path ORAM* (Stefanov et al., 2018)

- The server stores a binary tree of height $L = \lceil \lg n \rceil$ (level 0 is the root, level L are the leaves)

- Each node in the tree is called a "bucket", which stores Z blocks (real or dummy, and the server does not know which)
    - Typically $Z = 4$ or 5

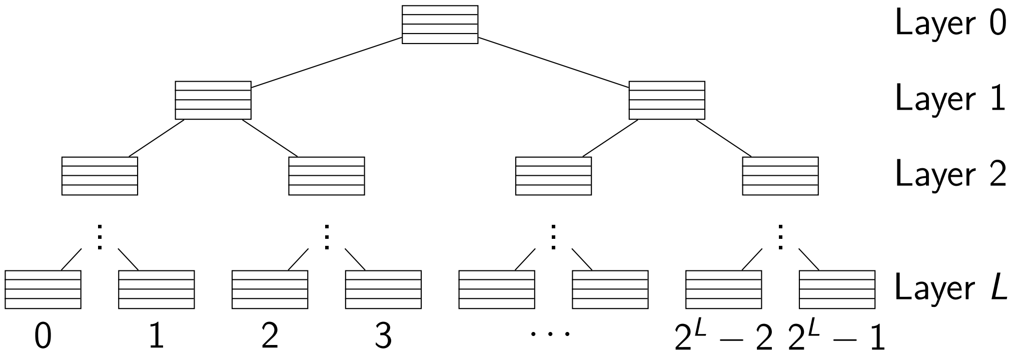
- The client has a "stash" of up to $\mathcal{O}(\lg n)$ blocks (not encrypted), and a "position map", which we'll talk about soon

Path ORAM



- The leaves are numbered consecutively
- The position map pos[$\cdot$] on the client maps block ids to *random* leaf numbers

Path ORAM



- Invariant: if pos[a] $=$ x, then block number a is either:
  1. in the stash
  2. somewhere on the path from the root to leaf x

Path ORAM

To access block a:

- Look up x $=$ pos[a]; assign a new random value to pos[a]
- Download the path from the root to leaf x into the stash
    - Downloading $Z(L+1)$ blocks
- Find block a in the stash, updating it if the access is a write
- Write back (reencrypted) buckets from leaf x to the root, putting as many real blocks as will fit and are allowed by the invariant into each bucket
    - If not all the real blocks can fit in the buckets, they stay in the stash
    - Uploading $Z(L+1)$ blocks

Client-server vs TEEs

- In the client-server setting, the server can see all accesses to its storage, but *cannot see the client's local computation at all*
- In the TEE setting, the adversarial OS can see all accesses to memory, *even those done by the client-side algorithm* (running in the TEE)
- So in the TEE setting, the "server" is just the untrusted OS, and the "client" is running inside the TEE
  - You need to write the client-side ORAM code itself in an oblivious manner!

One more step

- The client code in the TEE will need to access the stash and the position map obliviously
- The stash is pretty small ($\mathcal{O}(\lg n)$ entries)
    - Just use linear scan
- But the position map may be pretty large (n entries, but each is just a small number)
    - If the position map is not too large, linear scan is fine
    - Otherwise, you want to be able to read and update the position map without revealing which entry you're reading and updating

  $\Rightarrow$ Recursively use a smaller ORAM!

Building PIR from TEEs and ORAM

- Using a TEE-based ORAM (where the TEE is the single and only client of the ORAM), we can build a multi-client single-server PIR system for a database of r records of size s bytes each
- The server computation is only $\mathcal{O}(s \lg r)$ per query, instead of $\mathcal{O}(rs)$ for the PIR schemes we've seen so far

Building PIR from TEEs and ORAM

- Make each record in the PIR database be a block in the ORAM
  - So $n = r$ and $\beta = s$
- Clients (as many as you like) use remote attestation to establish a secure channel to the enclave in the TEE
  - Once per client, not once per PIR access
- The clients just send the record number they're looking for over the secure channel
- The ORAM client in the enclave looks up that block in the ORAM and returns it to the client over the secure channel

Bonus: private writes

- As a bonus, this mechanism also allows clients to *privately write* to the database at almost no extra cost:
- The client just sends (to the enclave in the TEE over the secure channel) the new value of the record along with the record number
- The enclave does an ORAM write instead of an ORAM read

Questions

- Why does this construction implement PIR?
  - Why can the server not tell which record is being accessed?
  - Why can the server not tell whether two accesses (by the same or different clients) are to the same record or different records?
- We said earlier that PIR schemes have to do $\mathcal{O}(rs)$ computation, because they have to access every record in the database
  - Otherwise, they learn that the client's desired record was not one of the untouched records
  - Why is this construction able to do better ($\mathcal{O}(s \lg r)$)?