Short Paper: Probabilistically Almost-Oblivious Computation

Ian Sweet,† David Darais★, Michael Hicks†
† University of Maryland ★ University of Vermont

ABSTRACT

Memory-trace Obliviousness (MTO) is a noninterference property: programs that enjoy it have neither explicit nor implicit information leaks, even when the adversary can observe the program counter and the address trace of memory accesses. *Probabilistic* MTO relaxes MTO to accept probabilistic programs. In prior work, we developed $\lambda_{\mathbf{obliv}}$, whose type system aims to enforce PMTO [2]. We showed that $\lambda_{\mathbf{obliv}}$ could typecheck (recursive) Tree ORAM [6], a sophisticated algorithm that implements a probabilistically oblivious key-value store. We conjectured that $\lambda_{\mathbf{obliv}}$ ought to be able to typecheck more optimized *oblivious data structures* (ODSs) [8], but that its type system was as yet too weak.

In this short paper we show we were wrong: *ODSs cannot be implemented in $\lambda_{\mathbf{obliv}}$ because they are not actually PMTO*, due to the possibility of *overflow*, which occurs when a `oram_write` silently fails due to a local lack of space. This was surprising to us because Tree ORAM can also overflow but is still PMTO. The paper explains what is going on and sketches the task of adapting the PMTO property, and $\lambda_{\mathbf{obliv}}$'s type system, to characterize ODS security.

CCS CONCEPTS

• **Security and privacy** → **Logic and verification**;

ACM Reference Format:
Ian Sweet,† David Darais★, Michael Hicks†. 2020. Short Paper: Probabilistically Almost-Oblivious Computation. In *15th Workshop on Programming Languages and Analysis for Security (PLAS '20), November 13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3411506.3417598

PLAS '20, November 13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8092-8/20/11…$15.00
https://doi.org/10.1145/3411506.3417598

1 ORAM AND TREE ORAM

An Oblivious RAM (ORAM) is a random-access memory, mapping (secret) `key`s to (secret) `data`.

```
type key   = nat<S>
type data  = ...
type oram  = ...
val oram_create : nat  → oram
val oram_read   : oram → key → bool<S> * data
val oram_write  : oram → key → data → bool<S>
```

We make keys natural numbers, for simplicity. The `<S>` annotation on types indicates they are considered secret. The `oram_read` operation returns a secret boolean indicating if the key was found in the ORAM. If this value is `false` then the data is garbage (a default value). Likewise, `oram_write` indicates whether or not the key and data were successfully written to the ORAM. This will only fail if the ORAM is full, so a `false` return value indicates *overflow*.

There are many deployment scenarios for ORAM but here is a simple one: A less-trusted server stores the data blocks, while a trusted client runs the ORAM code that retrieves these blocks. The client encrypts the data blocks (hiding their contents from the server) and it hides a block's relationship to its key in some way, e.g., by obfuscating the access pattern.

1.1 Trivial ORAM

The simplest ORAM implementation is *Trivial ORAM*, which is an array of key-value pairs, but with an extra bit indicating if the cell is occupied:

```
type oram = (bool<S> * key * data) array
```

For example, the Trivial ORAM `[| (true, 1, "a") ; (true, 3, "b") ; (true, 2, "c") ; (false, 0, "") |]` stores the value `"a"` at (logical) key `1`, `"c"` at `2`, and `"b"` at `3`. The last array cell is unoccupied, as indicated by the first component being `false`. Trivial ORAM's `oram_read` and `oram_write` operations access every address of the array; this way the (adversary-visible) address trace reveals nothing about whether the key is present in the ORAM or not. In our deployment scenario, the `oram` contents can all be stored server-side, while the code runs client-side. This code is, of course, inefficient: each operation takes time $O(n)$ where n is the size of the Trivial ORAM.

1.2 Tree ORAM

Modern ORAM implementations achieve performance $\Omega(\log(n))$ by employing *randomness* [3]. As an example, consider Tree ORAM [6]. Its memory is structured as a complete tree where each node (called a *bucket*) is a Trivial ORAM. Every `tree_read` performs all physical memory accesses along one particular path through the tree. Here is Tree ORAM's API and parts of its code:

```
type pos        = nat<S>
type cldata     = ...
type data       = pos * cldata
type tree_oram  = oram array

let default_block () = (false, 0, (rnd, ...))

```

```
8   val tree_create : nat → nat → tree_oram
9   let tree_create n m =
10    array[n](fun _ → array[m](fun _ → default_block ()))
11
12  val tree_read : tree_oram → key → nat → bool<S> * data
13  let tree_read t k p =
14    let len = length(t) in          (* len   = 2^k − 1, k > 0 *)
15    let depth = log 2 (len + 1) in  (* depth = k *)
16    let rec iterate level acc =     (* level goes from 0 ... k *)
17      if level = depth then acc
18      else
19        let base = (2 ** level) − 1 in
20        (* base + (p & base) is node on path p at level *)
21        let bucket = t[base + (p & base)] in
22        let curr = oram_read bucket k in
23        let (occupied, _, _) = curr in
24        let ret = mux occupied then curr else acc in
25        iterate (level + 1) ret
26    in
27    let (occupied, _, v) = iterate 0 (default_block ()) in
28    (occupied, v)
29
30  val tree_write : tree_oram → key → pos → cldata → bool<S>
31  let tree_write t k p v =
32    let overflow_write = oram_write t[0] k (p, v) in
33    let overflow_evict = evict t in
34    overflow_write || overflow_evict
```

This API differs from the `oram` API above in several respects; Tree ORAM can be used to implement full `oram`, as explained shortly. The type `pos` is a randomly generated natural number that acts as a *position tag*. Tree ORAM uses a position tag's binary representation to uniquely determine a path through the tree. The function `tree_create n m` creates a new Tree ORAM with Trivial ORAM buckets of size `m` and a number of nodes `n` (assumed to be $2^k - 1$ for some $k > 0$, ensuring a complete tree). Tree ORAM pairs up the position tag (type `pos`) of a block with the client data (type `cldata`) and store both in the underlying bucket. Figure 1 gives a graphical representation of the Tree ORAM produced by `tree_create 3 2` (on which we build an oblivious stack in Section 2). Here, each Trivial ORAM record is depicted *vertically*, with occupied bit `occupied`, key `key` and data `pos,next,data`.

The `tree_read t k p` function walks the path through the tree specified by the position tag, `p`, performing an `oram_read` at each node in search of the key `k`. The **mux** construct on line 24 is just like an **if** except that both branches are reduced to values before execution.[1] This ensures obliviousness when the guard is a secret.[2] Consider our deployment scenario: While the untrusted server does not know which block is returned, it does learn its position tag, based on the path taken; this is why the type of `tree_read`'s third argument is `nat`, not (secret-labeled) `pos`. We return to this point in the next section.

The `tree_write t k p v` simply writes the key, position tag, and value provided to the root node of the tree: `oram_write t[0] k (p, v)`. Afterwards, a procedure called `evict` is invoked to randomly "push" blocks down in the tree. The results in this paper hold for a variety of eviction procedures; here is a simple one: `evict` randomly chooses a block and pushes it down either left or right, depending on the position tag in that block (indeed, `evict` is the only reason that blocks include a position tag); `evict` also pushes a dummy block in the opposite direction. In this way, blocks are always stored along the appropriate path, but the adversary cannot tell which path that is from observing the memory trace.

[1]In Darais et al. [2], $\lambda_{\mathbf{obliv}}$ includes a two-component **mux** which produces an in-order tuple of both branches if the guard is `true`, swapping them otherwise. The one-component **mux** in this paper may be encoded by only binding the first component of the result of the two-component **mux**: `let (x, _) = mux(g, t, e) in ... .`

[2]For example, the following is unsafe: `if secret then (a[0]; ()) else ()`

We can implement a full `oram` as a pair `(o,m)` where `o` is a `tree_oram` and `m` is a *position map*, which maps keys to position tags. Because the position map is the same size as the Tree ORAM, the position map imposes an $O(n)$ space overhead, which in our deployment scenario is borne by the client. In particular, the Tree ORAM-based `oram_read (o,m) k` operation first looks up `m[k]` to retrieve tag `p` for `k` from the position map `m`; then it calls `tree_read o k p` to retrieve the value from the (server-side) Tree ORAM. An `oram_write (o,p) k v` generates a random position tag `p`, updates `m[k] = p`, and then calls `tree_write o k p v`. In fact, `oram_read` follows the call to `tree_read` with a call to `tree_write`, to put the value back in the ORAM at a fresh location. (`oram_write` may call `tree_read` before calling `tree_write`, to match the address trace of `oram_read`.) While the adversary can see the tag passed to `tree_read`, nothing is gleaned from it because it is never reused.

To reduce the client-side space cost of using Tree-based ORAM to a small constant, we can actually recursively store the position map across a sequence of $O(\log(b))$ Tree ORAMs, where b is the number of bits in a `pos`.

1.3 Probabilistic Memory Trace Obliviousness

Both Trivial and Tree ORAM enjoy *Probabilistic Memory Trace Obliviousness (PMTO)* [2]: for both, the distribution of adversary-visible events is independent of any secrets (the keys and values) they manipulate.

The type system of $\lambda_{\mathbf{obliv}}$ ensures that programs are PMTO by enforcing the invariant that random numbers revealed to the adversary are always uniformly distributed, conditioned on previously revealed random numbers. A random number is generated via the **rnd** expression, and is initially invisible to the adversary (like a `nat<S>`). The random number may be revealed to the potential adversary (i.e., made "public") *at most once*, enforced by the type system using affine types [4]. To add needed flexibility, a random number may also be coerced to a (normal) `nat<S>` number, which may be freely copied, but not revealed. To prevent such derived secret numbers from being used to perturb the uniformity of distributions of random numbers that have or will be revealed, the type system uses a feature called *probability regions*. The snippet below shows an example of a perturbation and how probability regions prevent it.

```
1  let sx, sy = flip, flip in
2  let sk = mux(castS(sx), sx, sy) in (* sk is non−uniform *)
3  let sz = mux(s, sk, flip) in
4  ...
```

In this example, `s` is a secret we wish to protect. The **flip** construct is exactly like **rnd** except that it produces a random boolean. Two random booleans, `sx` and `sy`, are created on line 1. On line 2, we use **castS** to coerce `sx`, which is of type **flip** into a `bool<S>`, so we can multiplex on it. Doing so will bind `sk` to either `sx` or `sy` depending on the value of `sx`. As such, `sk` is not uniformly distributed (it is more likely `true` than `false`). On line 3, we choose to bind `sz` to `sk` if `s` is `true` and a fresh, uniformly distributed boolean otherwise. If we were to reveal `sz`, the adversary could infer information about `s`; i.e., observing `true` means `s` is more likely to be `true` as well.

Probability regions in $\lambda_{\mathbf{obliv}}$ render a **mux** like the one that appears on line 2 as type-incorrect. On line 1, `sx` will be assigned some probability region ρ_1. Probability regions form a join semilattice which aligns with probabilistic (in)dependnce according to an ordering $\sqsubset$. On line 2, the type rule for **mux** checks that the region of the guard ρ_1 is strictly less than both of the arguments, meaning that they do not depend on it, probabilistically. In this case, since the left branch is sx, we require $\rho_1 \sqsubset \rho_1$ which does not hold. As such, $\lambda_{\mathbf{obliv}}$ will reject this program as unsafe. For all the juicy details of the $\lambda_{\mathbf{obliv}}$ type system and how it enforces the uniformity invariant, see Darais et al. [2].

The PMTO property holds for Trivial and Tree ORAM despite *overflow*. If a bucket fills up, a write to that bucket will have no effect, and a subsequent lookup will return the wrong answer. While undesirable, overflow is not observable by the adversary, and so PMTO of oram is not threatened. However, PMTO *is* compromised by overflow in *oblivious data structures*, as we describe next.

2 OBLIVIOUS DATA STRUCTURES

What if we wanted to implement an oblivious version of a data structure like a stack? For such a data structure, the visible address trace should reveal nothing about the data structure's contents nor anything about the operations being performed on it (e.g., which ones are pops vs. pushes). An easy way to do this is to store the structure's data in an oram, like a Tree ORAM, with a little meta-data stored client-side, e.g., the head key of the stack. To hide pushes vs. pops, one can (with a little effort) write the code to always perform the same sequence of ORAM operations, e.g., an oram_read always followed by an oram_write.

2.1 Tree ORAM-based Oblivious Data Structures

While using a full oram can work, it is space-inefficient: an oram of size n requires a position map of size n, even if the stack contains only a few elements. Wang et al. [8] proposed a clever way to reduce this overhead: Use a tree_oram, but replace the full ORAM's complete (size n) position map with one based on the data structure's API. We will generically refer to Wang et al.'s construction as an *oblivious data structure* (ODS). For oblivious stacks, we have:

```
1  type cldata = rnd * string<S>
2  type ostack = key ref * rnd ref * tree_oram
3
4  val empty : nat → nat → ostack
5  let empty n m = (ref 0, ref rnd, tree_create n m)
6
7  val stackop : ostack → bool<S> → string<S> → string<S>
8  let stackop (head_key, head_pos, stack) ispush v =
9    let hk = !head_key in
10   let hp = !head_pos in
11   if ispush then
12     (* Dummy read *)
13     let _ = tree_read stack 0 (castS rnd) in
14     let fresh = rnd in
15     let _ = tree_write stack hk (castS fresh) (hp, v) in
16     let () = head_key := hk + 1 in
17     let () = head_pos := fresh in
18     ""
19   else
20     let (_, (_, (next, v))) = tree_read stack (hk − 1) (castP hp) in
21     (* Dummy write *)
22     let _ = tree_write 0 (castS rnd) (rnd, "") in
23     let () = head_key := hk − 1 in
24     let () = head_pos := next in
25     v
```

An oblivious stack is a triple of a key, a (**rnd**) position tag, and a Tree ORAM. The first two components form a size-1 position map which points to the head of the stack (the only element a client

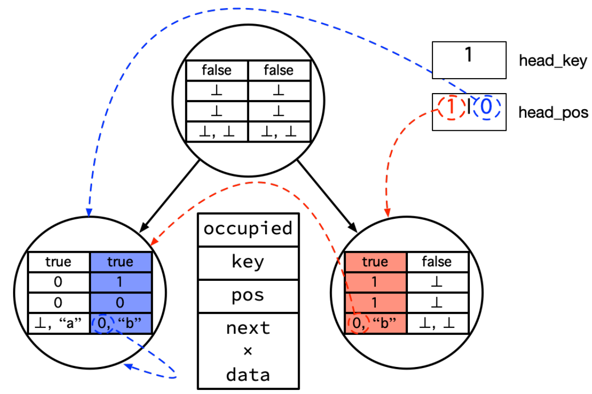

Figure 1: Visualizing an OStack after a push of "a" and then two possible outcomes (either blue or red) of a push of "b".

can access via the stack API); the head's key corresponds to the length of the stack (so it starts as 0). The position maps of the non-head stack elements are stored in the stack itself. In particular, type cldata contains the client's data in its second component, and the **rnd** component of the next element's position map in the first; the key is the current element's key, minus one. stackop takes a stack, a secret boolean indicating either push or pop (ispush), and some client data. If the operation is a push (ispush = true), stackop creates a new cldata object containing the pushed data and the current head's **rnd** tag ((hp,v) on line 15). It then calls tree_write with a fresh position tag and new key to add the new object to the tree_oram; the new key is the old head's key plus one. Finally, it updates the current head to contain the new key and tag. If the operation is a pop, stackop looks up the head and returns the client data but also the pointer to the next element in the stack (next on line 20), which becomes the new head. The implementation of stackop ignores the overflow bit returned by tree_write. Doing so matches the behavior described by Wang et al. [8], which (we assume) aims to make an overflow adversary-*in*visible, thereby preserving PMTO. As we show in this section, ignoring overflow actually does the opposite, i.e., it compromises PMTO. The stackop code uses an **if** expression for clarity. Since the ispush variable is considered secret, a real implementation would need to **mux** instead. See Darais et al. [2] for a full description of stackop (including the version that uses **mux**), and pseudocode.

Figure 1 shows the configuration of an ODS stack after two pushes. The pair head_key,head_pos are the pointer to the head of the stack (we depict the position tag as either 1 or 0 since the figure considers two possible executions for the second push; see below). Each block in the Tree ORAM has the usual fields: the occupied bit occupied, the key key, position tag pos, and client data cldata. The first push generates a fresh position tag, which happens to be 0. We add the block (true, head_key, 0, (head_pos, "a")) = (true, 0, 0, (⊥, "a")) to the Tree ORAM,[3] and it is evicted left because its tag is 0. The head_key is incremented, and head_pos is updated to 0. An identical procedure describes the second push, but in Figure 1 we instead show both possible outcomes for the fresh, random position tag, p. Blue indicates the outcome p = 0 and red indicates p = 1. We add the block (true, head_key + 1, p, (head_pos, "b")) = (true, 1, p, (0, "b")).

[3] Here, ⊥ represents a garbage next pointer, since there is no next element.

α	β	$Pr(\rho = 0 \mid \gamma = 0)$	$Pr(\rho = 1 \mid \gamma = 0)$
0	0	0.5	0.5
0	1	0	1
1	0	1	0
1	1	0	1

Figure 2: Distribution of ρ conditioned on $\gamma = 0$.

The dashed arrows in Figure 1 indicate the bucket to which the associated key and position tag refer, revealing the abstract linked-list structure.

2.2 Tree ORAM-based Stack is not PMTO

We would expect ODSs to enjoy PMTO because the underlying Tree ORAM is PMTO and ODS operations can be made oblivious. We were surprised to find that this is not the case! The reason owes to the possibility of overflow in the Tree ORAM. If we were to implement a stack on top of a *full* Tree ORAM, with a complete position map, overflow will compromise correctness but not security. But for an ODS, some of the stack's metadata—in particular, the `next` pointers to neighboring elements—is stored *inside* the Tree ORAM, and that metadata can be corrupted on an overflow in a way that affects the adversary-visible address trace.

To see how, consider the blue configuration in Figure 1. This Tree ORAM configuration results from pushing `"a"` and `"b"` onto the stack with position tags α and β respectively, with $\alpha = \beta = 0$. The Trivial ORAM associated with the left child is full. Consider the unlucky situation in which the value `"c"` is pushed onto the stack with a generated position tag, γ, of `0`. The `head_key` and `head_pos` are updated to `2` and `0` respectively but the block containing `"c"` is not added to the underlying Tree ORAM due to overflow. If a pop operation is executed, γ is revealed to the adversary and the position tag, ρ, will be returned to the client. Under most executions, the client will be returned `"c"` and the returned position tag will be $\rho = \beta$. However, in the overflowing execution, the client will instead receive garbage. The returned position tag is $\rho = \delta$ where δ is some fresh, uniform position tag.

Figure 2 shows the distribution of ρ conditioned on the observation that $\gamma = 0$. For PMTO to hold, this distribution marginalized over α and β needs to be uniform. In the first row, we see the overflow case. In this case, $\rho = \delta$ and since δ is a fresh, uniform tag we see that ρ is zero or one with equal probability. In all other cases, $\rho = \beta$. Since the outcome of γ does not affect the probability distributions of α or β, each row in the table occurs with probability $\frac{1}{4}$. Therefore, when we marginalize over α and β we have $Pr(\rho = 1 \mid \gamma = 0) = \frac{5}{8}$ and $Pr(\rho = 0 \mid \gamma = 0) = \frac{3}{8}$. When the next pop takes place, ρ will also be revealed to the adversary (again, via `tree_read`), since it is assumed by the oblivious stack to be the position tag of `b`. If the adversary observes $\gamma = 0$ and $\rho = 1$ (say), they know that it is (slightly) more likely that an overflow took place. This observation of overflow leaks information about the operations being performed on the data structure, which are considered secret.

3 EXTENDING PMTO, AND $\lambda_{\mathbf{obliv}}$

While $\lambda_{\mathbf{obliv}}$'s type system accepts Trivial ORAM, Tree ORAM, and recursive Tree ORAM—and thereby establishes they are PMTO—a $\lambda_{\mathbf{obliv}}$ ODS stack fails to type check. We had previously thought [2] this was due to a weakness in $\lambda_{\mathbf{obliv}}$'s type system, but now it is clear that the rejection is warranted: ODS stacks are not PMTO.

While ODSs do not enjoy PMTO, they *almost* do—if an ODS does not overflow, it should satisfy PMTO. As such, one can reduce the chances of a leak by sizing the ODS to be close to its client's *working set size*. Moreover, using a tree-based ORAM like *Path ORAM* [7], which employs a kind of client-side cache, we can reduce the chances of overflow still further. Both ideas are discussed in the original ODS paper [8] (but not the problems with overflow).

We would like to formalize the idea of *PMTO modulo overflow* and extend $\lambda_{\mathbf{obliv}}$'s type system to enforce it. To do so, we would add allowable *declassifications* [5] to $\lambda_{\mathbf{obliv}}$ which would be used to declassify the result of the `oram_write` operations (which detect overflow). The PMTO modulo overflow property is very similar to *gradual release* [1], but lifted to distributions. Implementing this in $\lambda_{\mathbf{obliv}}$'s type system will be challenging. The actual overflow is not the place in the code where the type checker currently fails (which relates to the problem of creating a correlation between random variables). We also wonder: what does "overflow" mean, for general code (i.e., not ORAM)? Going further, we would like to extend the new PMTO property and the type system with the ability to reason *quantitatively* about the chances of overflow, and thus connect the "statistical closeness" of an ODS's distribution to the uniform distribution of events. Doing so will require reasoning about correctness properties of Trivial ORAM and Tree (or Path) ORAM with regards to overflow.

ACKNOWLEDGMENTS

We thank Kesha Hietala and the anonymous reviewers for comments on earlier drafts of this paper, and Elaine Shi for helpful discussions throughout our process. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1563722 and CCF-1901278; by ODNI/IARPA via 2019-1902070008. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF, ODNI, IARPA, or the U.S. Government.

REFERENCES

[1] Aslan Askarov and Andrei Sabelfeld. 2007. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP '07)*. IEEE Computer Society, USA, 207–221. https://doi.org/10.1109/SP.2007.22

[2] David Darais, Ian Sweet, Chang Liu, and Michael Hicks. 2019. A Language for Probabilistically Oblivious Computation. *Proc. ACM Program. Lang.* 4, POPL, Article 50 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371118

[3] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *J. ACM* 43, 3 (May 1996).

[4] Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages*. The MIT Press.

[5] Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and Principles. *J. Comput. Secur.* 17, 5 (Oct. 2009).

[6] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O((logN)3) Worst-Case Cost. In *ASIACRYPT*. 197–214.

[7] Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *J. ACM* 65, 4, Article 18 (April 2018).

[8] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In *CCS*.