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 \( \lambda_{\text{obliv}} \), whose type system aims to enforce PMTO [2]. We showed that \( \lambda_{\text{obliv}} \) could typecheck (recursive) Tree ORAM [6], a sophisticated algorithm that implements a probabilistically oblivious key-value store. We conjectured that \( \lambda_{\text{obliv}} \) 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 \( \lambda_{\text{obliv}} \) because they are not actually PMTO, due to the possibility of overflow, which occurs when a \texttt{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 \( \lambda_{\text{obliv}} \)'s type system, to characterize ODS security.

CCS CONCEPTS

- Security and privacy → Logic and verification;

ACM Reference Format:


1 ORAM AND TREE ORAM

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

\begin{verbatim}
1 type key = nat>S
2 type data = ...
3 type oram = ...
4 val oram_create : nat → oram
5 val oram_read : oram → key → bool>S + data
6 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 \texttt{oram_read} operation returns a secret boolean indicating if the key was found in the ORAM. If this value is \texttt{false} then the data is garbage (a default value). Likewise, \texttt{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 \texttt{false} return value indicates \texttt{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 \textit{Trivial ORAM}, which is an array of key-value pairs, but with an extra bit indicating if the cell is occupied:

\begin{verbatim}
1 type oram = (bool>S * key * data) array
\end{verbatim}

For example, the Trivial ORAM \([| (true , 1, "a") ; (true , 3, "b") ; (false , 2, "c") |] \) 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 \texttt{false}. Trivial ORAM’s \texttt{oram_read} and \texttt{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 \texttt{oram} contents can all be stored server-side, while the code runs client-side. This code is, of course, inefficient: each operation takes time \( \Omega(n) \) where \( n \) is the size of the Trivial ORAM.

1.2 Tree ORAM

Modern ORAM implementations achieve performance \( \Omega(\log(n)) \) by employing \textit{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 \texttt{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:

\begin{verbatim}
1 type pos = nat>S
2 type cldata = ...
3 type data = pos * cldata
4 type tree_oram = oram array
5
6 let default_block () = (false , 0, (rnd, ...))
\end{verbatim}
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 \(k \geq 0\), 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], \(\lambda_{obliv}\) 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 (s, _) = mux(g, t, e) in ...\n
\(^2\)For example, the following is unsafe: if secret then (a [0], {}) else ()

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 \(\lambda_{obliv}\) 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 \(\text{rand}\) 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.

\begin{verbatim}
let sx, sy = flip, flip in
let sk = mux(cast(sx), sx, sy) in (* sk is non-uniform *)
let sz = mux(s, sk, flip) in
\end{verbatim}

In this example, s is a secret we wish to protect. The flip construct is exactly like \(\text{rand}\) except that it produces a random boolean. Two random booleans, \(sx\) and \(sy\), are created on line 1. On line 2, we use \text{cast}\ 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 \(\lambda_{obliv}\) ensure a mux like the one that appears on line 2 as type-incorrect. On line 1, sx will be assigned some probability region \(p_1\). Probability regions form a join semilattice.
which aligns with probabilistic (in)dependence according to an ordering □. On line 2, the type rule for \texttt{mux} checks that the region of the guard \( p_1 \) 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 \( \alpha \), we require \( p_1 \sqsubseteq p_1 \) which does not hold. As such, \( \lambda \text{obliv} \) will reject this program as unsafe. For all the juicy details of the \( \lambda \text{obliv} \) 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 \texttt{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 \texttt{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 \texttt{oram_read} always followed by an \texttt{oram_write}.

2.1 Tree ORAM-based Oblivious Data Structures

While using a full \texttt{oram} can work, it is space-inefficient: an \texttt{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 \texttt{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 \textit{oblivious data structure} (ODS). For oblivious stacks, we have:

```latex
\begin{verbatim}
1  \textbf{type cldata = rnd} * string<S>
2  \textbf{type stack} = \texttt{key ref} * \texttt{rnd ref} * \texttt{tree_oram}
3  \textbf{val empty : nat \to nat \to \texttt{stack}}
4  \textbf{let empty n m = (\texttt{ref 0, ref \texttt{ rnd}, \texttt{tree_create n m)}})
5  \textbf{val stackop : \texttt{stack} \to \texttt{bool<5>} \to \texttt{string<5>} \to \texttt{string<5>}}
6  \textbf{let stackop (head_key, head_pos, stack) ispush v =}
7    \textbf{let hk = 'head_key in}
8    \textbf{let hp = 'head_pos in}
9    \textbf{if ispush then}
10      \langle 'Dummy read- \rangle
11    \textbf{else}
12      \langle 'Mutex read \rangle
13      \textbf{let } l := \texttt{tree_read stack 0 (castS rnd) in}
14      \textbf{let fresh := rnd in}
15      \textbf{let } \texttt{h} := \texttt{tree_write stack hk (castS fresh) (hp, v) in}
16      \textbf{let } () := \texttt{head_key := hk + 1 in}
17      \textbf{let } () := \texttt{head_pos := fresh in}
18
19\end{verbatim}
```

An oblivious stack is a triple of a key, a \texttt{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 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 \texttt{cldata} contains the client’s data in its second component, and the \texttt{rnd} component of the next element’s position map in the first; the key is the current element’s key, minus one. \texttt{stackop} takes a stack, a secret boolean indicating either push or pop (\texttt{ispush}), and some client data. If the operation is a push (\texttt{ispush = true}), \texttt{stackop} creates a new \texttt{cldata} object containing the pushed data and the current head’s \texttt{rnd} tag ((\texttt{hp,v}) on line 15). It then calls \texttt{tree_write} with a fresh position tag and new key to add the new object to the \texttt{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, \texttt{stackop} looks up the head and returns the client data but also the pointer to the next element in the stack (\texttt{next} on line 20), which becomes the new head. The implementation of \texttt{stackop} ignores the overflow bit returned by \texttt{tree_write}. Doing so matches the behavior described by Wang et al. [8], which (we assume) aims to make an overflow adversary-invisible, thereby preserving PMTO. As we show in this section, ignoring overflow actually does the opposite, i.e., it compromises PMTO. The \texttt{stackop} code uses an if expression for clarity. Since the \texttt{ispush} variable is considered secret, a real implementation would need to \texttt{mux} instead. See Darais et al. [2] for a full description of \texttt{stackop} (including the version that uses \texttt{mux}), and pseudocode.

Figure 1 shows the configuration of an ODS stack after two pushes. The pair \texttt{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 \texttt{occupied}, the key \texttt{key}, position tag \texttt{pos}, and client data \texttt{cldata}. The first push generates a fresh position tag, which happens to be 0. We add the block (\texttt{true, head_key, 0, (head_pos, 'a')}) = (\texttt{true, 0, 0, ('a')}) to the Tree ORAM, and it is evicted left because its tag is 0. The \texttt{head_key} is incremented, and \texttt{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, \texttt{p}. Blue indicates the outcome \texttt{p = 0} and red indicates \texttt{p = 1}. We add the block (\texttt{true, head_key + 1, p, (head_pos, 'b')}) = (\texttt{true, 1, p, (0, 'b')}).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Visualizing an OStack after a push of ‘a’ and then two possible outcomes (either blue or red) of a push of ‘b’.}
\end{figure}

\footnote{Here, \( \perp \) represents a garbage \texttt{next} pointer, since there is no next element.}
While \(\lambda_{\text{obliv}}\)’s type system accepts Trivial ORAM, Tree ORAM, and recursive Tree ORAM—and thereby establishes they are PMTO—a \(\lambda_{\text{obliv}}\) ODS stack fails to type check. We had previously thought [2] this was due to a weakness in \(\lambda_{\text{obliv}}\)’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 \(\lambda_{\text{obliv}}\)’s type system to enforce it. To do so, we would add allowable declassifications [5] to \(\lambda_{\text{obliv}}\), 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 \(\lambda_{\text{obliv}}\)’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 19-1902070008. 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**


<table>
<thead>
<tr>
<th>(\alpha)</th>
<th>(\beta)</th>
<th>(Pr(\rho = 0))</th>
<th>(Pr(\rho = 1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2: Distribution of \(\rho\) conditioned on \(y = 0\).