A Language for Probabilistically Oblivious Computation

DAVID DARAIS, University of Vermont
IAN SWEET, University of Maryland
CHANG LIU, Citadel Securities
MICHAEL HICKS, University of Maryland

An oblivious computation is one that is free of direct and indirect information leaks, e.g., due to observable differences in timing and memory access patterns. This paper presents $\lambda_{\textbf{obliv}}$, a core language whose type system enforces obliviousness. Prior work on type-enforced oblivious computation has focused on deterministic programs. $\lambda_{\textbf{obliv}}$ is new in its consideration of programs that implement *probabilistic* algorithms, such as those involved in cryptography. $\lambda_{\textbf{obliv}}$ employs a substructural type system and a novel notion of *probability region* to ensure that information is not leaked via the distribution of visible events. Probability regions support reasoning about *probabilistic correlation* between values, and our use of probability regions is motivated by a source of unsoundness that we discovered in the type system of ObliVM, a language for implementing state of the art oblivious algorithms. We prove that $\lambda_{\textbf{obliv}}$'s type system enforces obliviousness and show that it is powerful enough to check advanced tree-based oblivious RAMs.

1 INTRODUCTION

Cloud computing allows clients to conveniently outsource computation, but they must trust that cloud providers do not exploit or mishandle sensitive information. To remove the provider from the trusted computing base, work in both industry and research has strived to produce a secure abstract machine comprising an execution engine and protected memory: The adversary cannot see sensitive data as it is being operated on, nor can it observe such data at rest in memory. Such an abstract machine can be realized by encrypting the data in memory and then performing computations using cryptographic mechanisms (e.g., secure multi-party computation [Yao 1986]) or secure processors [Hoekstra 2015; Suh et al. 2003; Thekkath et al. 2000].

Unfortunately, a secure abstract machine does not defend against an adversary that can observe memory access patterns [Islam et al. 2012; Maas et al. 2013; Zhuang et al. 2004] and instruction timing [Brumley and Boneh 2003; Kocher 1996], among other "side" channels of information. For cloud computing, such an adversary is the cloud provider itself, which has physical access to its machines, and so can observe traffic on the memory bus.

A countermeasure against an unscrupulous provider is to augment the secure processor to store code and data in *oblivious RAM* (ORAM) [Maas et al. 2013; Suh et al. 2003]. First proposed by Goldreich and Ostrovsky [Goldreich 1987; Goldreich and Ostrovsky 1996], ORAM obfuscates the mapping between addresses and data, in effect "encrypting" the addresses along with the data. Replacing RAM with ORAM solves (much of) the security problem but incurs a substantial slowdown in practical situations [Liu et al. 2015a, 2013; Maas et al. 2013] as reads/writes add overhead that is polylogarithmic in the size of the memory.

Recent work has explored methods for reducing the cost of programming with ORAM. Liu et al. [2015a, 2013, 2014] developed a family of type systems to check when *partial* use of ORAM (alongside normal, encrypted RAM) results in no loss of security; i.e., only when the addresses of secret data could (indirectly) reveal sensitive information must the data be stored in ORAM. This optimization can provide order-of-magnitude (and asymptotic) performance improvements. Wang et al. [2014] explored how to build *oblivious data structures* (ODSs), such as queues or stacks,

Authors' addresses: David Darais, University of Vermont, david.darais@uvm.edu; Ian Sweet, University of Maryland, ins@cs.umd.edu; Chang Liu, Citadel Securities, liuchang@eecs.berkeley.edu; Michael Hicks, University of Maryland, mwh@cs.umd.edu.

that are more efficient than their standard counterparts implemented on top of ORAM. Their technique involves specializing ideas from ORAM algorithms to particular data structures, resulting in asymptotic performance gains in common cases. In followup work, Liu et al. [2015b] devised a programming language called ObliVM[1] for implementing such oblivious data structures, and ORAMs themselves. While Liu et al.'s earlier work treats ORAM as a black box, in ObliVM one can program ORAM algorithms as well as ODSs. A key feature of ObliVM is careful treatment of random numbers, which are at the heart of state-of-the-art ORAM and ODS algorithms. While the goal of ObliVM is that well-typed programs are secure, no formal argument to this effect is made.

In this paper, we present $\lambda_{\textbf{obliv}}$, a core language for oblivious computation, inspired by ObliVM. $\lambda_{\textbf{obliv}}$ extends a core language equipped with higher order functions, primitives for generating and using uniformly distributed random numbers, and a novel type system which in part resembles an information flow type system [Sabelfeld and Myers 2006]. We prove that $\lambda_{\textbf{obliv}}$'s type system guarantees *probabilistic memory trace obliviousness* (PMTO), i.e., that the possible distribution of adversary-visible execution traces is independent of the values of secret variables. This property generalizes the deterministic MTO property enforced by Liu et al. [2015a, 2013], which did not consider the use of randomness. In carrying out this work, we discovered that the ObliVM type system is unsound, so an important contribution of $\lambda_{\textbf{obliv}}$ is to address the problem without overly restricting or complicating the language.

$\lambda_{\textbf{obliv}}$'s type system aims to ensure that no probabilistic correlation forms between secrets and publicly revealed random choices. In oblivious algorithms it is often the case that a security-sensitive random choice is made (e.g., where to store a particular block in an ORAM), and eventually that choice is made visible to the adversary (e.g., when a block is accessed by the client). This transition from a secret value to a public one—which we call a *revelation*—is not problematic so long as the revealed value does not communicate information about a secret. $\lambda_{\textbf{obliv}}$ ensures that revelations do not communicate information by guaranteeing that all revealed values are uniformly distributed.

$\lambda_{\textbf{obliv}}$'s type system, presented in Section 3, ensures that revelations are uniformly distributed by treating randomly generated numbers as *affine*, meaning they cannot be freely copied. This prohibition prevents revealing the same number twice, which would be problematic if allowed because the second revelation is not uniformly distributed. Unfortunately, strict affinity is too strong for implementing oblivious algorithms, which require the ability to make copies of random numbers which are later revealed. $\lambda_{\textbf{obliv}}$'s type system addresses this by allowing random numbers to be copied as non-affine secret values which can never be revealed. Moreover, $\lambda_{\textbf{obliv}}$ enforces that random numbers do not influence the choice of whether or not they are revealed, since this could also result in a non-uniform revelation. For example, a $\lambda_{\textbf{obliv}}$ program cannot copy a random number to a secret, look at that secret, and then decide whether or not to reveal the original random number. The type system prevents this problem by using a new mechanism we call *probability regions* to track the probabilistic (in)dependence of values in the program. (Probability regions are missing in ObliVM, and their absence is the source of ObliVM's unsoundness.) Section 4 outlines the proof that $\lambda_{\textbf{obliv}}$ enjoys PMTO by relating its semantics to a novel *mixed semantics* whose terms operate on distributions directly, which makes it easier to state and prove the PMTO property.

While we have not retrofitted $\lambda_{\textbf{obliv}}$'s type system into ObliVM, we have implemented a type checker for an extension of $\lambda_{\textbf{obliv}}$. Section 5 presents an implementation of a tree-based ORAM, a state-of-the-art class of ORAM implementations [Shi et al. 2011; Stefanov et al. 2013; Wang et al. 2015], that our language type checks. We also show, in Appendix A.1, that we can type check *oblivious stacks*, a kind of oblivious data structure [Wang et al. 2014], in $\lambda_{\textbf{obliv}}$, though with some caveats. As far as we are aware, our implementations constitute the first automated "proofs" that

[1] http://www.oblivm.com

these data structures are indeed oblivious. Though further improvements can be made, $\lambda_{\textbf{obliv}}$'s support for randomness strictly generalizes prior work on obliviousness by typing; section 6 discusses related work.

2 OVERVIEW

This section presents the definition of the threat model and background on type-enforced deterministic oblivious execution. The next section motivates and sketches our novel type system for enforcing probabilistic oblivious execution.

2.1 Threat Model

We assume a powerful adversary that can make fine-grained observations about a program's execution. In particular, we use a generalization of the *program counter (PC) security model* [Molnar et al. 2006]: The adversary knows the program being executed, can observe the PC during execution as well as the contents and patterns of memory accesses. Some *secret* memory contents may be encrypted (while *public* memory is not) but all memory addresses are still visible.

As a relevant instantiation, consider an untrusted cloud provider using a secure processor, like SGX [Hoekstra 2015]. Reads/writes to/from memory can be directly observed, but secret memory is encrypted (using a key kept by the processor). The pattern of accesses, timing information, and other system features (e.g., instruction cache misses) provide information about the PC. Another instantiation is secure multi-party computation (MPC) using secret shares [Goldreich et al. 1987]. Here, two parties simultaneously execute the same program (and thus know the program and program counter), but certain values, once entered by one party or the other, are kept hidden from both using secret sharing.

Our techniques can also handle weaker adversaries, such as those that can observe memory traffic but not the PC, or can make timing measurements but cannot observe the PC or memory.

2.2 Oblivious Execution

Our goal is to ensure *memory trace obliviousness (MTO)*, which is a kind of noninterference property [Goguen and Meseguer 1982; Sabelfeld and Myers 2006]. This property states that despite being able to observe each address (of instructions and data) as it is fetched, and each public value, the adversary will not be able to infer anything about input secret values.

We can formalize this idea as a small-step operational semantics $\sigma; e \longrightarrow^{t} \sigma'; e'$, which states that an expression e in memory σ transitions to memory σ' and expression e' while emitting trace event t. Trace events include fetched instruction addresses, public values, and the addresses of public and secret values that are read and written. (Secret *values* are not visible in the trace.) Under this model, the MTO property means that running *low-equivalent* programs $\sigma_1; e_1$ and $\sigma_2; e_2$—meaning they agree on the code and public values but may not agree on secret ones—will produce the exact same memory trace, along with low-equivalent results. More formally, if $\sigma_1; e_1 \sim \sigma_2; e_2$ then $\sigma_1; e_1 \longrightarrow^{t_1} \sigma_1'; e_1'$ and $\sigma_2; e_2 \longrightarrow^{t_2} \sigma_2'; e_2'$ imply $t_1 = t_2$ and $\sigma_1'; e_1' \sim \sigma_2'; e_2'$, where operator $\sim$ denotes low-equivalence.

To illustrate how revealing addresses can leak information, consider the program in Figure 1(a). Here, we assume array `B`'s contents are secret, and thus invisible to the adversary. Variables `s0`, `s1`, and `s` are secret (i.e., encrypted) inputs. The assignments on the first two lines are safe since we are storing secret values in the secret array. The problem is on the last line, when the program uses `s` to index `B`. Since the adversary is able to see which addresses are accessed (in address trace t), they are able to infer `s`.

The program in Figure 1(b) fixes the problem. It reads both secret values from `B`, and then uses the **mux** to select the one indicated by `s`, storing it in `r`. The semantics of **mux** is that if the first

```
1  B[0] ← s0
2  B[1] ← s1
3   ...
4  let s = ... // secret bit
5  let r = B[s] // leaks s
```

(a) Leaky program

```
1  B[0] ← s0
2  B[1] ← s1
3   ...
4  let s = ... // secret bit
5  let s0' = B[0]
6  let s1' = B[1]
7  let r,_ = mux(s,s1',s0')
```

(b) Deterministic MTO program

```
1  let sk = flip ()
2  let s0', s1' = mux(castS(sk),s1,s0)
3  B[0] ← s0'
4  B[1] ← s1'
5   ...
6  let s = ... // secret bit
7  let s' = xor(s,sk)
8  let r = B[castP(s')]
```

(c) Probabilistic MTO program

Fig. 1. Code examples

argument is 1 it pairs and returns the second two arguments in order, otherwise it swaps them. To the adversary this appears as a single program instruction , and so nothing is learned about s via branching. Moreover, nothing is learned from the address trace: We always unconditionally read both elements of B, no matter the value of s.

While this approach is secure, it is inefficient: To read a single secret value in B this code reads *all* values in B, to hide which one is being selected. If B were an array of size N, this approach would turn an $O(1)$ operation into an $O(N)$ operation.

2.3 Probabilistic Oblivious Execution

To improve performance while retaining security, the key is to employ *randomness*. In particular, the client can randomly generate and hold secret a key, using it to map logical addresses used by the program to physical addresses visible to the adversary. We can illustrate this approach with the program in Figure 1(c), which hints at the basic approach to implementing an ORAM. Rather than deterministically store s0 and s1 in positions 0 and 1 of B, respectively, the program scrambles their locations according to a coin flip, sk, generated by the call to **flip**, and not visible to the adversary. Using the **mux** on line 2, if sk is 1 then s0 and s1 will be copied to s0' and s1', respectively, but if sk is 0 then s0 and s1 will be swapped, with s0 going into s1' and s1 going into s0'. (The **castS** coercion on sk is a no-op, used by the type system; it will be explained in the next subsection.) Values s0' and s1' are then stored at positions 0 and 1, respectively, on lines 3 and 4. When the program later wishes to look up the value at logical index s, it must consult sk to retrieve the mapping. This is done via the **xor** on line 7. Then s' is used to index B and retrieve the value logically indicated by s.

In terms of memory accesses, this program is more efficient: It is reading B only once, rather than twice. One can argue that more work is done overall, but as we will see in Section 5, this basic idea does scale up to build full ORAMs with access times of $O(log_c\ N)$ for some c (rather than $O(N)$).

This program is also secure: no matter the value of s, the adversary can learn nothing from the address trace. Consider the Figure 2 which categorizes the four possible traces (the memory indexes used to access B) depending on the possible values of s and sk. This table makes plain that our program is not *deterministically* MTO. Looking at column sk=0, we can see that a program that has s=0 may produce trace 0,1,0 while a program that uses s=1 may produce trace 0,1,1; MTO programs may not produce different traces when using different secrets.

But this is not actually a problem. Assuming that sk = 0 and sk = 1 are equally likely, we can see that address traces 0,1,0 and 0,1,1 are also equally likely no matter whether s = 0 or s = 1. More specifically, if we assume the adversary's expectation for secret values is uniformly distributed, then after *conditioning* on knowledge of the third memory access, the adversary's expectation for the secret remains unchanged, and thus nothing is learned about s. This probabilistic model of adversary knowledge is captured by a *probabilistic* variant of MTO. In particular, the probability of any particular trace event t emitted by two low-equivalent

	sk=0	sk=1
s=0	0,1,0	0,1,1
s=1	0,1,1	0,1,0

Fig. 2. Possible traces

```
1  let sx,sy = (flip (), flip ())
2  let sz,_ = mux (s,sx,sy)
3  output (castP(sz)) (* OK *)
4  output (castP(sx)) (* Bad *)
```

(a) Leak by multiple revelation

```
1  let sx,sy = (flip (), flip ())
2  let sk,_ = mux(castS(sx),sx,sy)
3  let sz,_ = mux(s,sk,flip ())
4  output (castP(sz)) (* Bad *)
```

(b) Leak due to probabilistic dependence

Fig. 3. Example leaky programs (precluded by type system)

programs should be the same for both programs, and the resulting programs should also be low-equivalent. More formally: If $\sigma_1; e_1 \sim \sigma_2; e_2$ then $\Pr[\sigma_1; e_1 \longrightarrow^t \sigma_1'; e_1'] = q_1$ and $\Pr[\sigma_2; e_2 \longrightarrow^t \sigma_2'; e_2'] = q_2$ implies $q_1 = q_2$ and $\sigma_1'; e_1' \sim \sigma_2'; e_2'$ when $q_i > 0$.

2.4 $\lambda_{\textbf{obliv}}$: Obliviousness by Typing

The main contribution of this paper is $\lambda_{\textbf{obliv}}$, a language and type system design such that well-typed programs are probabilistically MTO, and an expressive computation model that supports examples like those shown in Figure 1(c). Unlike prior work [Liu et al. 2015a, 2013], $\lambda_{\textbf{obliv}}$ does not assume the presence of an ORAM as a black box. Rather, as Section 5 shows, $\lambda_{\textbf{obliv}}$'s type system is powerful enough to type check ORAM implementations, which use randomness to improve efficiency [Shi et al. 2011; Stefanov et al. 2013; Wang et al. 2015]. $\lambda_{\textbf{obliv}}$ is also powerful enough to type check oblivious algorithms beyond ORAMs, as discussed in Appendix A.1.

$\lambda_{\textbf{obliv}}$'s type system's power derives from two key features: *affine* treatment of random values, and *probability regions* to track probabilistic (in)dependence (i.e., correlation) between random values that could leak information when a value is revealed. Together, these features ensure that each time a random value is revealed to the adversary—even if the value interacted with secrets, like the secret memory layout of an ORAM—it is *always uniformly distributed*, which means that its particular value communicates no secret information.

Affinity. In $\lambda_{\textbf{obliv}}$, public and secret bits are given types `bitP` and `bitS` respectively, and coin flips are given type `flip`. (Our formalism uses bits for simplicity; it is easy to generalize to (random) integers, which is done in our implementation.) Values of `flip` type are, like secret bits of type `bitS`, invisible to the adversary. But a `flip` can be revealed by using `castP` to convert it to a public bit, as is done on line 8 of Figure 1(c) to perform a (publicly visible) array index operation.

The type system aims to ensure that a `flip` value is always uniformly distributed when it is revealed. The uniformity requirement implies that each flip should be revealed *at most once.* Why? Because the second time a flip is revealed, its distribution is conditioned on prior revelations, meaning the each outcome is no longer equally likely. To see how this situation could end up leaking secret information, consider the example in Figure 3(a). Lines 1–3 in this code are safe: we generate two coin flips that are invisible to the adversary, and then store one of them in `sz` depending on whether the secret `s` is 1 or not. Revealing `sz` at line 3 is safe: regardless of whether `sz` contains the contents of `sx` or `sy`, the fact that both are uniformly distributed means that whatever is revealed, nothing can be learned about `s`. However, revealing `sx` on line 4, after having revealed `sz`, is not safe. This is because seeing two ones or two zeroes in a row is more likely when `sz` is `sx`, which happens when `s` is one. So this program violates PMTO.

To prevent this problem, $\lambda_{\textbf{obliv}}$'s type system treats values of type `flip` *affinely*, meaning that each can be used at most once. The read of `sx` on line 2 consumes that variable, so it cannot be used again on the problematic line 4. Likewise, flip variable `sk` is consumed when passed to **xor** on line 7 of Figure 1(c), and `s'` is consumed when revealed on line 8.

Unfortunately, a purely affine treatment of flips would preclude useful algorithms. In particular, notice that line 2 of Figure 1(c) uses `sk` as the guard of a **mux**. If doing so consumed `sk`, line 7's use

of sk would fail to type check. To avoid this problem, $\lambda_{\textbf{obliv}}$ relaxes the affinity constraint on flips passed to **castS**. In effect, programs can make many secret $\texttt{bit}_S$ copies of a coin, and compute with them, but only the original flip can ultimately be revealed.

It turns out that this relaxed treatment of affinity is insufficient to ensure PMTO (and is the source of the bug in ObliVM [Liu et al. 2015b]). The reason is that we can now use non-affine copies of a coin to make a flip's distribution non-uniform when it is revealed. To see how, consider the code in Figure 3(b). This code flips two coins, and then uses the **mux** to store the first coin flip, sx, in sk if sx is 1, else to store the second coin flip there. Now sk is more likely to be 1 than not: $\Pr[\texttt{sk} = 1] = \frac{3}{4}$ while $\Pr[\texttt{sk} = 0] = \frac{1}{4}$. On line 3, the **mux** will store sk in sz if secret s is 1, which means that if the adversary observes a 1 from the output on line 4, it is more likely than not that s is 1. The same sort of issue would happen if we replaced line 1 from Figure 1(c) with the first two lines above: when the program looks up B[**castP**(s')] on line 8, if the adversary observes 1 for the address, it is more likely that s is 0, and vice versa if the adversary observes 1. Notice that we have not violated affinity here: no coin flip has been used more than once (other than uses of **castS** which side-step affinity tracking).

Probability regions. $\lambda_{\textbf{obliv}}$'s type system addresses the problem of probabilistic correlations leading to non-uniform distributions using a novel construct we call *probability regions*. Both bit and **flip** types are ascribed to a region ρ, which represents a set of coin flips (reminiscent of a points-to location in alias analysis [Emami et al. 1994]). We have elided the region name in our examples so far, but normally should write $\texttt{flip}^{\rho}()$ for flipping a coin in region ρ, which then has type $\texttt{flip}^{\rho}$. We should also write $\texttt{bit}^{\rho}_{S}$ and $\texttt{bit}^{\rho}_{P}$ for secret and public bit types.

Regions are organized as a join semi-lattice. The type system enforces an invariant that each flip labeled with region ρ is probabilistically independent of all bits derived from flips at regions ρ' when $\rho' \sqsubset \rho$. Normal bits (i.e., those not produced via **castS**) will have region $\bot$. (We elide region annotation from the bit type when ρ is $\bot$.) Bits derived from flips via **castS** carry the region of the original flip. Then, the type system will prevent problematic correlations arising among bits and flips, in particular via the **mux** and **xor** operations, in a way that could threaten uniformity.

We can see regions at work in the problematic example above: the region of the secret bit **castS**(sx) is the same region as sx, since **castS**(sx) was derived from sx. As such, there is no assurance of probabilistic independence between the guard and the branch; indeed, when conditioning on **castS**(sx) to return sx, the output will not be uniform. On the other hand, if the guard of a **mux** is a bit in region ρ and its branches are flips in region ρ' where $\rho \sqsubset \rho'$, then the guard is derived from a flip that is sure to be independent of the branches, so the uniformity of the output is not threatened.

3 FORMALISM

This section presents the syntax, semantics, and type system of $\lambda_{\textbf{obliv}}$. The following section proves that $\lambda_{\textbf{obliv}}$'s type system is sufficient to ensure PMTO.

3.1 Syntax

Figure 4 shows the syntax for $\lambda_{\textbf{obliv}}$. The term language is expressions e. The set of values v is comprised of (1) base values such as variables x (included to enable a substitution-based semantics) and recursive function definitions $\texttt{fun}_y(x{:}\tau).e$ where the function body may refer to itself using variable y; and (2) connectives from the expression language e which identify a subset of expressions which are also values, such as pairs $\langle v, v\rangle$ with type $\tau \times \tau$.

Expressions also include bit literals b_ℓ (of type $\texttt{bit}^{\bot}_{\ell}$) which are either 0 or 1 and annotated with their security label ℓ. Bit literals are not values in order to distinguish them from those that appear at runtime; we introduce bit values separately for the "standard" semantics and "mixed" as they are

$\ell \in \text{label} ::= \mathsf{P} \mid \mathsf{S}$		public and secret
	$(\textit{where } \mathsf{P} \sqsubset \mathsf{S})$	security labels
$\rho \in R ::= \ldots$		probability region
$b \in \mathbb{B} ::= \mathtt{0} \mid \mathtt{I}$		bits
$x, y \in \text{var} ::= \ldots$		variables
$v \in \text{val} ::= x$		variable values
$\mid \mathtt{fun}_y(x{:}\tau).e$		function values
$\mid \langle v, v \rangle$		tuple values
$\tau \in \text{type} ::= \mathtt{bit}^{\rho}_{\ell}$		non-random bit
$\mid \mathtt{flip}^{\rho}$		secret uniform bit
$\mid \mathtt{ref}(\tau)$		reference
$\mid \tau \times \tau$		tuple
$\mid \tau \to \tau$		function

$e \in \text{exp} ::= v$	value expressions
$\mid b_{\ell}$	bit literal
$\mid \mathtt{flip}^{\rho}()$	coin flip in region
$\mid \mathtt{cast}_{\ell}(v)$	cast flip to bit
$\mid \mathtt{mux}(e, e, e)$	atomic conditional
$\mid \mathtt{xor}(e, e)$	bit xor
$\mid \mathtt{if}(e)\{e\}\{e\}$	branch conditional
$\mid \mathtt{ref}(e)$	reference creation
$\mid \mathtt{read}(e)$	reference read
$\mid \mathtt{write}(e, e)$	reference write
$\mid \langle e, e \rangle$	tuple creation
$\mid \mathtt{let}\ x = e\ \mathtt{in}\ e$	variable binding
$\mid \mathtt{let}\ x, y = e\ \mathtt{in}\ e$	tuple elimination
$\mid e(e)$	fun. application

Fig. 4. $\lambda_{\textbf{obliv}}$ Syntax (source programs)

treated differently in each. A security label ℓ is either S (secret) or P (public). Values with the former label are invisible to the adversary. Bit types include this security label along with a probability region ρ. The expression $\mathtt{flip}^{\rho}()$ produces a coin flip, i.e., a randomly generated bit of type $\mathtt{flip}^{\rho}$. The annotation assigns the coin to region ρ which may not be $\bot$. Coin flips are semantically secret, and have limited use; we can compute on one using $\mathtt{mux}$ or $\mathtt{xor}$, cast one to a public bit via $\mathtt{cast}_{\mathsf{P}}$, or cast to a secret bit via $\mathtt{cast}_{\mathsf{S}}$. To simplify the type system, casts only apply to values, however $\mathtt{cast}_{\ell}(e)$ could be used as shorthand for $\mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{cast}_{\ell}(x)$.

The expression $\mathtt{mux}(e_1, e_2, e_3)$ unconditionally evaluates e_2 and e_3 and returns their values as a pair in the given order if e evaluates to $\mathtt{I}$, or in the opposite order if it evaluates to $\mathtt{0}$. This operation is critical for obliviousness because it is atomic. By contrast, normal conditionals $\mathtt{if}(e_1)\{e_2\}\{e_3\}$ evaluate either e_2 or e_3 depending on e_1, never both, so the branch taken is evident directly in the trace. The components of tuples e_1 constructed as $\langle e_1, e_2 \rangle$ can be accessed via $\mathtt{let}\ x_1, x_2 = e_1\ \mathtt{in}\ e_2$. $\lambda_{\textbf{obliv}}$ also has normal let binding, function application, and means to create mutable reference cells with read and write operations.

$\lambda_{\textbf{obliv}}$ captures the key elements that make implementing oblivious algorithms possible, notably: random and secret bits, trace-oblivious multiplexing, public revelation of secret random values, and general computational support in tuples, conditionals and recursive functions. Other features can be encoded in these, e.g., general numbers and operators on them can be encoded as tuples of bits, and arrays can be encoded as tuples of references, read/written using (nested) conditionals. Our prototype interpreter implements these things directly, as well as region polymorphism (Section 5).

3.2 Semantics

Figure 5 presents a monadic, probabilistic small-step semantics for $\lambda_{\textbf{obliv}}$ programs. The top of the figure contains some new and extended syntax. Values (and, by extension, expressions) are extended with forms for bit values $\mathtt{bitv}_{\ell}(b)$, flip values $\mathtt{flipv}(b)$, and reference locations $\mathtt{locv}(\iota)$; these do not appear in source programs. Stores σ map locations to values. Stores are paired with expressions to form *configurations* ς. A sequence of configurations arising during an evaluation is collected in a *trace* t. We use Hieb-Felleisen-style contexts [Felleisen and Hieb 1992] to define E (not shown) which follows a left-to-right, call-by-value evaluation strategy.

The semantics is defined using an abstract probability monad $\mathcal{M}$. Below the semantics we define the standard "denotational" probability monad $\mathcal{D}$ [Giry 1982; Ramsey and Pfeffer 2002], where $\mathcal{D}(A)$ represents a probability distribution over objects in some countable set A. The "standard"

$$
\begin{array}{llll}
\iota \in \mathrm{loc} \approx \mathbb{N} & \text{ref locations} & \sigma \in \mathrm{store} \triangleq \mathrm{loc} \rightharpoonup \mathrm{val} & \text{store} \\
v \in \mathrm{val} ::= \ldots & \text{extended...} & e \in \mathrm{exp} ::= \ldots & \text{extended...} \\
\quad \mid \mathtt{bitv}_\ell(b) & \text{bit value} & \varsigma \in \mathrm{config} ::= \sigma, e & \text{configuration} \\
\quad \mid \mathtt{flipv}(b) & \text{uniform bit value} & t \in \mathrm{trace} ::= \epsilon \mid t \cdot \varsigma & \text{trace} \\
\quad \mid \mathtt{locv}(\iota) & \text{location value} & E \in \mathrm{context} ::= \ldots & \text{eval contexts...}
\end{array}
$$

$$\boxed{\mathrm{step}_{\mathcal{M}} \in \mathbb{N} \times \mathrm{config} \to \mathcal{M}(\mathrm{config})}$$

$$
\begin{array}{ll}
\mathrm{step}_{\mathcal{M}}(N, \sigma, b_\ell) & = \mathrm{return}(\sigma, \mathtt{bitv}_\ell(b)) \\
\mathrm{step}_{\mathcal{M}}(N, \sigma, \mathtt{flip}^\rho()) & = \mathrm{do}\ b \leftarrow \mathrm{bit}(N)\ ;\ \mathrm{return}(\sigma, \mathtt{flipv}(b)) \\
\mathrm{step}_{\mathcal{M}}(N, \sigma, \mathtt{cast}_\ell(\mathtt{flipv}(b))) & = \mathrm{return}(\sigma, \mathtt{bitv}_\ell(b)) \\
\mathrm{step}_{\mathcal{M}}(N, \sigma, \mathtt{mux}(\mathtt{bitv}_{\ell_1}(b_1), \mathtt{bitv}_{\ell_2}(b_2), \mathtt{bitv}_{\ell_3}(b_3))) & = \mathrm{return}(\sigma, \langle \mathtt{bitv}_\ell(b_2'), \mathtt{bitv}_\ell(b_3') \rangle) \\
\quad \textit{where}\quad b_2' \triangleq \mathrm{cond}(b_1, b_2, b_3) \quad \textit{and}\quad b_3' \triangleq \mathrm{cond}(b_1, b_3, b_2) \quad \textit{and}\quad \ell \triangleq \ell_1 \sqcup \ell_2 \sqcup \ell_3 & \\
\mathrm{step}_{\mathcal{M}}(N, \sigma, \mathtt{mux}(\mathtt{bitv}_\ell(b_1), \mathtt{flipv}(b_2), \mathtt{flipv}(b_3))) & = \mathrm{return}(\sigma, \langle \mathtt{flipv}(b_2'), \mathtt{flipv}(b_3') \rangle) \\
\quad \textit{where}\quad b_2' \triangleq \mathrm{cond}(b_1, b_2, b_3) \quad \textit{and}\quad b_3' \triangleq \mathrm{cond}(b_1, b_3, b_2) & \\
\mathrm{step}_{\mathcal{M}}(N, \sigma, \mathtt{if}(\mathtt{bitv}_\ell(b))\{e_1\}\{e_2\}) & = \mathrm{return}(\sigma, \mathrm{cond}(b, e_1, e_2)) \\
\mathrm{step}_{\mathcal{M}}(N, \sigma, \mathtt{xor}(\mathtt{bitv}_\ell(b_1), \mathtt{flipv}(b_2))) & = \mathrm{return}(\sigma, \mathtt{flipv}(b_1 \oplus b_2)) \\
\mathrm{step}_{\mathcal{M}}(N, \sigma, \mathtt{ref}(v)) & = \mathrm{return}(\sigma[\iota \mapsto v], \mathtt{refv}(\iota)) \quad \textit{where}\ \iota \notin \mathrm{dom}(\sigma) \\
\mathrm{step}_{\mathcal{M}}(N, \sigma, \mathtt{read}(\mathtt{refv}(\iota))) & = \mathrm{return}(\sigma, \sigma(\iota)) \\
\mathrm{step}_{\mathcal{M}}(N, \sigma, \mathtt{write}(\mathtt{refv}(\iota), v)) & = \mathrm{return}(\sigma[\iota \mapsto v], \sigma(\iota)) \\
\mathrm{step}_{\mathcal{M}}(N, \sigma, \mathtt{let}\ x = v\ \mathtt{in}\ e) & = \mathrm{return}(\sigma, [v/x]e) \\
\mathrm{step}_{\mathcal{M}}(N, \sigma, \mathtt{let}\ x_1, x_2 = \langle v_1, v_2 \rangle\ \mathtt{in}\ e) & = \mathrm{return}(\sigma, [v_1/x_1][v_2/x_2]e) \\
\mathrm{step}_{\mathcal{M}}(N, \sigma, (\underbrace{\mathtt{fun}_y(x : \tau).\ e}_{v_1})(v_2)) & = \mathrm{return}(\sigma, [v_1/y][v_2/x]e) \\
\mathrm{step}_{\mathcal{M}}(N, \sigma, E[e]) & = \mathrm{do}\ \sigma', e' \leftarrow \mathrm{step}_{\mathcal{M}}(N, \sigma, e)\ ;\ \mathrm{return}(\sigma', E[e']) \\
\mathrm{step}_{\mathcal{M}}(N, \sigma, v) & = \mathrm{return}(\sigma, v)
\end{array}
$$

$$\boxed{\mathrm{nstep}_{\mathcal{M}} \in \mathbb{N} \times \mathrm{config} \to \mathcal{M}(\mathrm{trace})}$$

$$
\begin{array}{l}
\mathrm{nstep}_{\mathcal{M}}(0, \varsigma) = \mathrm{return}(\epsilon \cdot \varsigma) \\
\mathrm{nstep}_{\mathcal{M}}(N+1, \varsigma) = \mathrm{do}\ t \cdot \varsigma' \leftarrow \mathrm{nstep}_{\mathcal{M}}(N, \varsigma)\ ;\ \varsigma'' \leftarrow \mathrm{step}_{\mathcal{M}}(N+1, \varsigma')\ ;\ \mathrm{return}(t \cdot \varsigma' \cdot \varsigma'')
\end{array}
$$

$$\tilde{x} \in \mathcal{D}(A) \triangleq \left\{ f \in A \to \mathbb{R} \;\middle|\; \sum_{x \in A} f(x) = 1 \right\} \qquad \Pr[\tilde{x} \doteq x] \triangleq \tilde{x}(x) \qquad \boxed{\mathcal{D}(A) \in \mathrm{set}}$$

$$
\begin{array}{ll}
\mathrm{return} \in \mathcal{D}(A) & \mathrm{return}(x) \triangleq \lambda x'.\ \begin{cases} 1 & \textit{if}\quad x = x' \\ 0 & \textit{if}\quad x \neq x' \end{cases} \\
\mathrm{bind} \in \mathcal{D}(A) \times (A \to \mathcal{D}(B)) \to \mathcal{D}(B) & \mathrm{bind}(\tilde{x}, f) \triangleq \lambda y.\ \sum_x f(x)(y)\tilde{x}(x) \\
\mathrm{bit} \in \mathbb{N} \to \mathcal{D}(\mathbb{B}) & \mathrm{bit}(N) \triangleq \lambda b.\ 1/2
\end{array}
$$

Fig. 5. $\lambda_{\mathbf{obliv}}$ Semantics

semantics for our language occurs when $\mathcal{M} = \mathcal{D}$, and we leave $\mathcal{M}$ a parameter so we can instantiate the semantics to a new monad in later sections.

In the probability monad $\mathcal{D}$, the return operation constructs a point distribution, and the bind operation encodes the law of total probability, i.e., constructs a marginal distribution from a conditional one. We only use proper distributions in the sense that the combined mass of all

elements sums to 1. We do not denote possibly non-terminating programs directly into the monad, and therefore do not require the use of domains [?] or sub-distributions [?]—we use the monad only to denote distributions of configurations which occur after some finite number of small-step transitions, which is always well-defined.

The definition of $\text{step}_{\mathcal{M}}$ describes how a single configuration advances in a single probabilistic step, yielding a distribution of resulting configurations. The definition uses Haskell-style do notation as the usual notation for bind. Starting from the bottom, we can see that a value v advances to itself (more on why, below) and evaluating a redex e within a context E steps the former and packages its result back with the latter, as usual. The cases for let binding, pair deconstruction, and function application are standard, using a substitution-based semantics. Likewise, rules for creating, reading, and writing from references operate on the store σ as usual.

Moving to the first case, we see that literals b_ℓ evaluate in one step to bit values. A $\texttt{flip}^\rho()$ expression evaluates to either a $\texttt{flipv}(\texttt{I})$ or $\texttt{flipv}(\texttt{O})$ as determined by sampling from the input tape via $\text{bit}(N)$. The monad $\mathcal{D}$ does not use the N parameter in its definition of $\text{bit}(N)$, however a later monad will. The $\texttt{cast}_\ell$ case converts a coin to a similarly-labeled bit value. The next few cases use the three-argument metafunction $\text{cond}(b, X, Y)$, which returns X if b is bit $\texttt{I}$, and Y otherwise. The two **mux** cases operate in a similar way: they return the second two arguments of the **mux** in order when the first argument is $\texttt{bitv}_\ell(\texttt{I})$, and in reverse order when it is $\texttt{bitv}_\ell(\texttt{O})$. The security label of the result is the join of the labels of all elements in involved. This is not needed for flip values, since these are always fixed to be secret. The case for **if** also uses cond in the expected manner. The case for **xor** permits xor-ing a bit with a coin, returning a coin.

The bottom of the figure defines function $\text{nstep}_{\mathcal{M}}(N, \varsigma)$. It composes N invocations of $\text{step}_{\mathcal{M}}$ starting at ς to produce a distribution of traces t.

3.3 Type System

Figure 6 defines the type system for $\lambda_{\textbf{obliv}}$ source programs as rules for judgment $\Gamma \vdash e : \tau \;;\; \Gamma'$, which states that under type environment Γ expression e has type τ, and yields residual type environment Γ'. We discuss typing configurations, including non-source program values, in the next section. Type environments map variables to either types τ or inaccessibility tags $\bullet$. The latter are used to enforce that values of $\texttt{flip}^\rho$ type are *affine* in typestate style. As discussed in Section 2.4, the type system does this to help ensure that whenever a coin flip is publicly revealed, its distribution always uniform. It also enforces the invariant that flips at *probability region* ρ are always probabilistically independent of bits derived from flips at regions ρ' when $\rho' \sqsubset \rho$. Finally, the type system employs standard mechanisms to prevent *information leaks*. We discuss the type rules organized around these concepts.

Affinity. To enforce non-duplicability, when an affine variable is used by the program, its type is removed from the residual environment. Figure 6 defines kinding metafunction $\mathcal{K}$ that assigns a type either the kind universal $\texttt{U}$ (freely duplicatable) or affine $\texttt{A}$ (non-duplicatable). Bits, functions, and references (but not their contents, necessarily) are always universal, and flips are always affine. A pair is considered affine if either of its components is. Rule VarU in Figure 6 types universally-kinded variables; the output environment Γ is the same as the input environment. Rule VarA types an affine variable by marking it $\bullet$ in the output environment. This rule is sufficient to rule out the first problematic example in Section 2.4.

Rules Cast-S and Cast-P permit converting $\texttt{flip}^\rho$ types to bits via the $\texttt{cast}_\texttt{S}$ and $\texttt{cast}_\texttt{P}$ coercions, respectively. The first converts a $\texttt{flip}^\rho$ to a $\texttt{bit}^\rho_\texttt{S}$ and does *not* make its argument inaccessible (it returns the original Γ) while the second converts to a $\texttt{bit}^\rho_\texttt{P}$ and does make it inaccessible (returning

$\overset{\bullet}{\tau} \in \overset{\bullet}{\text{type}} ::= \tau \mid \bullet \quad (\textit{where } \tau \sqsubset \bullet)$
$\kappa \in \text{kind} ::= \mathtt{U} \mid \mathtt{A} \quad (\textit{where } \mathtt{U} \sqsubset \mathtt{A})$

$\Gamma \in \text{tcxt} \triangleq \mathit{var} \rightharpoonup \overset{\bullet}{\text{type}}$
$(\Gamma_1 \sqcup \Gamma_2)(x) \triangleq \Gamma_1(x) \sqcup \Gamma_2(x)$

$\boxed{\mathcal{K} \in \text{type} \to \text{kind}}$

$\mathcal{K}(\mathtt{bit}^{\rho}_{\ell}) \triangleq \mathcal{K}(\tau_1 \to \tau_2) \triangleq \mathcal{K}(\mathtt{ref}(\tau)) \triangleq \mathtt{U} \qquad \mathcal{K}(\mathtt{flip}^{\rho}) \triangleq \mathtt{A} \qquad \mathcal{K}(\tau_1 \times \tau_2) \triangleq \mathcal{K}(\tau_1) \sqcup \mathcal{K}(\tau_2)$

$\boxed{\Gamma \vdash e : \tau \;;\; \Gamma}$

VarU
$$\frac{\mathcal{K}(\Gamma(x)) = \mathtt{U} \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau \;;\; \Gamma}$$

VarA
$$\frac{\mathcal{K}(\Gamma(x)) = \mathtt{A} \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau \;;\; \Gamma[x \mapsto \bullet]}$$

Bit
$$\frac{}{\Gamma \vdash b_{\ell} : \mathtt{bit}^{\bot}_{\ell} \;;\; \Gamma}$$

Flip
$$\frac{}{\Gamma \vdash \mathtt{flip}^{\rho}() : \mathtt{flip}^{\rho} \;;\; \Gamma}$$

Cast-S
$$\frac{\Gamma \vdash x : \mathtt{flip}^{\rho} \;;\; \_}{\Gamma \vdash \mathtt{cast}_{\mathrm{S}}(x) : \mathtt{bit}^{\rho}_{\mathrm{S}} \;;\; \Gamma}$$

Cast-P
$$\frac{\Gamma \vdash x : \mathtt{flip}^{\rho} \;;\; \Gamma'}{\Gamma \vdash \mathtt{cast}_{\mathrm{P}}(x) : \mathtt{bit}^{\bot}_{\mathrm{P}} \;;\; \Gamma'}$$

If
$$\frac{\Gamma \vdash e : \mathtt{bit}^{\bot}_{\mathrm{P}} \;;\; \Gamma' \qquad \begin{array}{c}\Gamma' \vdash e_1 : \tau \;;\; \Gamma_1'' \\ \Gamma' \vdash e_2 : \tau \;;\; \Gamma_2''\end{array}}{\Gamma \vdash \mathtt{if}(e)\{e_1\}\{e_2\} : \tau \;;\; \Gamma_1'' \sqcup \Gamma_2''}$$

Mux-Bit
$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathtt{bit}^{\rho_1}_{\ell_1} \;;\; \Gamma' \\ \Gamma' \vdash e_2 : \mathtt{bit}^{\rho_2}_{\ell_2} \;;\; \Gamma'' \qquad \ell = \ell_1 \sqcup \ell_2 \sqcup \ell_3 \\ \Gamma'' \vdash e_3 : \mathtt{bit}^{\rho_3}_{\ell_3} \;;\; \Gamma''' \qquad \rho = \rho_1 \sqcup \rho_2 \sqcup \rho_3\end{array}}{\Gamma \vdash \mathtt{mux}(e_1, e_2, e_3) : \mathtt{bit}^{\rho}_{\ell} \times \mathtt{bit}^{\rho}_{\ell} \;;\; \Gamma'''}$$

Mux-Flip
$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathtt{bit}^{\rho_1}_{\ell_1} \;;\; \Gamma' \qquad \rho_1 \sqsubset \rho_2 \\ \Gamma' \vdash e_2 : \mathtt{flip}^{\rho_2} \;;\; \Gamma'' \qquad \rho_1 \sqsubset \rho_3 \\ \Gamma'' \vdash e_3 : \mathtt{flip}^{\rho_3} \;;\; \Gamma''' \qquad \rho = \rho_1 \sqcup \rho_2 \sqcup \rho_3\end{array}}{\Gamma \vdash \mathtt{mux}(e_1, e_2, e_3) : \mathtt{flip}^{\rho} \times \mathtt{flip}^{\rho} \;;\; \Gamma'''}$$

Xor-Flip
$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathtt{bit}^{\rho_1}_{\ell_1} \;;\; \Gamma' \\ \Gamma' \vdash e_2 : \mathtt{flip}^{\rho_2} \;;\; \Gamma'' \qquad \rho_1 \sqsubset \rho_2\end{array}}{\Gamma \vdash \mathtt{xor}(e_1, e_2) : \mathtt{flip}^{\rho_2} \;;\; \Gamma''}$$

Ref
$$\frac{\Gamma \vdash e : \tau \;;\; \Gamma'}{\Gamma \vdash \mathtt{ref}(e) : \mathtt{ref}(\tau) \;;\; \Gamma'}$$

Read
$$\frac{\mathcal{K}(\tau) = \mathtt{U} \qquad \Gamma \vdash e : \mathtt{ref}(\tau) \;;\; \Gamma'}{\Gamma \vdash \mathtt{read}(e) : \tau \;;\; \Gamma'}$$

Write
$$\frac{\Gamma \vdash e_1 : \mathtt{ref}(\tau) \;;\; \Gamma' \qquad \Gamma' \vdash e_2 : \tau \;;\; \Gamma''}{\Gamma \vdash \mathtt{write}(e_1, e_2) : \tau \;;\; \Gamma''}$$

Tup
$$\frac{\Gamma \vdash e_1 : \tau_1 \;;\; \Gamma' \qquad \Gamma' \vdash e_2 : \tau_2 \;;\; \Gamma''}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \;;\; \Gamma''}$$

Fun
$$\frac{\Gamma^{+} = \Gamma \uplus [x \mapsto \tau_1, y \mapsto (\tau_1 \to \tau_2)] \qquad \Gamma^{+} \vdash e : \tau_2 \;;\; \Gamma^{+\prime} \qquad \Gamma^{+\prime} = \Gamma \uplus [x \mapsto \_, y \mapsto \_]}{\Gamma \vdash \mathtt{fun}_y(x : \tau_1).\ e : \tau_1 \to \tau_2 \;;\; \Gamma}$$

App
$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \;;\; \Gamma' \qquad \Gamma' \vdash e_2 : \tau_1 \;;\; \Gamma''}{\Gamma \vdash e_1(e_2) : \tau_2 \;;\; \Gamma''}$$

Let
$$\frac{\Gamma \vdash e_1 : \tau_1 \;;\; \Gamma' \qquad \Gamma'^{+} = \Gamma' \uplus [x \mapsto \tau_1] \qquad \Gamma'^{+} \vdash e_2 : \tau_2 \;;\; \Gamma''^{+} \qquad \Gamma''^{+} = \Gamma'' \uplus [x \mapsto \_]}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2 \;;\; \Gamma''}$$

Let-Tup
$$\frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2 \;;\; \Gamma' \qquad \Gamma'^{+} = \Gamma' \uplus [x_1 \mapsto \tau_1, x_2 \mapsto \tau_2] \qquad \Gamma'^{+} \vdash e_2 : \tau_3 \;;\; \Gamma''^{+} \qquad \Gamma''^{+} = \Gamma'' \uplus [x_1 \mapsto \_, x_2 \mapsto \_]}{\Gamma \vdash \mathtt{let}\ x_1, x_2 = e_1\ \mathtt{in}\ e_2 : \tau_3 \;;\; \Gamma''}$$

Fig. 6. $\lambda_{\mathbf{obliv}}$ Type System (source programs)

Γ'). In essence, the type system is enforcing that any random number is made adversary-visible at most once; secret copies are allowed because they are never revealed.

References may contain affine values, but references themselves are universal. Rather than track the affinity of aliased contents specifically, the Read rule disallows reading out of a reference cell

whose contents are affine. Since the write operation returns the *old* contents of the cell, programs can see the existing contents of any reference by first writing in a valid replacement. This pattern is in the style of Linear LISP [Baker 1992], and is permitted by the Write rule. Its impact on programmability is discussed in Section 5.

The Fun rule ensures that no affine variables in the defining context are consumed within the body of the function, i.e., they are not captured by its closure. We write $\Gamma \uplus [x \mapsto \_, y \mapsto \_]$ to split a context into the part that binds x and y and Γ binds the rest; it is the Γ part that is returned, dropping the x and y bindings. Both Let and Let-Tup similarly remove their bound variables.

Finally, note that different variables could be made inaccessible in different branches of a conditional, so If types each branch in the same initial context, but then joins their the output contexts (as defined at the top of Figure 6); if a variable is made inaccessible by one branch, it will be inaccessible in the joined environment.

Information flow. The type system aims to ensure that bits b_ℓ whose security label ℓ is secret S cannot be learned by an adversary. Bit types $\texttt{bit}^\rho_\ell$ include the security label ℓ. The rules treat types with different labels as distinct, preventing so-called *explicit* flows. For example, the Write rule prevents assigning a secret bit (of type $\texttt{bit}^\rho_\mathsf{S}$) to a reference whose type is $\texttt{ref}(\texttt{bit}^\rho_\mathsf{P})$. Likewise, a function of type $\texttt{bit}^\rho_\mathsf{P} \to \tau$ cannot be called with an argument of type $\texttt{bit}^\rho_\mathsf{S}$, per the App rule. In our implementation we relax App (but not Write, due to the invariance of reference types) to allow public bits when secrets are expected; this is not done here just to keep things simpler.

The rules also aim to prevent *implicit* information flows. A typical static information flow type system [Sabelfeld and Myers 2006] would require the type of the conditional's guard to be less secret than the type of what it returns; e.g., the guard's type could be $\texttt{bit}^\rho_\mathsf{S}$ but only if the final type τ is secret too. However, in $\lambda_{\textbf{obliv}}$ we must be even more restrictive: rule If requires the guard bit value to have label P since the adversary-visible execution trace reveals which branch is taken, and thus the truth of the guard. Branching on secrets must be done via **mux**. Notice that rule Mux-Bit sets the label ℓ of the each element of the returned pair to be the join of the labels on the guard and the remaining components. As such, if the guard was secret, then the returned results will necessarily be. The Mux-Flip rule always returns coins, which are invisible to the adversary, so the guard can be either secret or public.

Probability regions. A *probability region* ρ appears on both `bit` and `flip` types. The region is a static name for a collection of flip values and secret bit values that may be derived from them. A flip value is associated with a region ρ when it is created, per rule Flip. Rule Cast-S ascribes the region ρ from the input $\texttt{flip}^\rho$ to the output type $\texttt{bit}^\rho_\mathsf{S}$, tracking the flip value(s) from which the secret bit value was possibly derived. Per rule Bit, bit literals have probability region $\bot$, as do public bits produced by $\texttt{cast}_\mathsf{P}$, per rule Cast-P.

The type system maintains the invariant that flips at region ρ are probabilistically independent of all secret bits in regions ρ' when $\rho' \sqsubset \rho$. This independence property is enforced when typing Mux-Flip rule. If a secret bit is typed at region ρ_1 and a flip value at region ρ_2, and $\rho_1 \not\sqsubset \rho_2$, then it may be that the values are correlated, and a `mux` involving the values may produce flips that are non-uniform. The strict ordering $\sqsubset$ is irreflexive; when a flip value is converted via $\texttt{cast}_\mathsf{S}$ the resulting bit maintains the same region. Since this region is $\not\sqsubset$ to that of the flip it was derived from, the problematic `mux` in Figure 3 (b) line 2 is untypable. Both the Mux-Flip and Mux-Bit rules return outputs whose region is the join of the regions of all inputs, indicating that the result of the `mux` is only independent of values that were jointly independent of each component of the `mux`.

These rules ensure the problematic example from the end of Section 2.4 is rejected, as it could produce a non-uniform coin sk. We recast the example below, labeled (a), using a pair of regions $\rho_1 \sqsubset \rho_2$.

```
1  let sx,sy = (flip^ρ1 () , flip^ρ2 () )
2  let sk,_ = mux(castS(sx),sx,sy)
```

(a) Incorrect example

```
1  let sx = flip^ρ1 () in
2  let sy,sz = mux(castS(sx),flip^ρ2 () , flip^ρ2 () )
```

(b) Correct example

The type checker first ascribes types $\texttt{flip}^{\rho_1}$ and $\texttt{flip}^{\rho_2}$ to sx and sy, respectively, according to rules Let-Tup, Flip, and Tup. It uses Cast-S to give castS(sx) type $\texttt{bit}_S^{\rho_1}$ and leaves sx accessible so that VarA can be used to give it and sy types $\texttt{flip}^{\rho_1}$ and $\texttt{flip}^{\rho_2}$, respectively (then making them inaccessible). Rule Mux-Flip will now fail because the independence conditions do not hold. In particular, the region ρ_1 of the guard is not strictly less than the region ρ_1 of the second argument, i.e., $\rho_1 \sqsubset \rho_1$ does not hold. On the other hand, the program labeled (b) above is well-typed. Here, the Mux-Flip rule is happy because the bit in the guard has region ρ_1, the region of the two flips is ρ_2 and $\rho_1 \sqsubset \rho_2$ as required. It is easy to see that regardless of sx's value, both sy and sz are uniformly distributed and independent of sx.

Rule Xor-Flip permits xor'ing a secret with a coin, returning a coin, as long as the secret's region and the coin's region are well ordered, which preserves uniformity. Rule Mux-Bit requires no ordering checks—the output bits inherit the (join of the) region(s) of the arguments.

An earlier version of the $\lambda_{\textbf{obliv}}$ type system did not order regions, but rather attempted to simply maintain an invariant that flips and bits in distinct regions are independent. This turns out to not work. While at the outset a fresh flip value is independent of all other values in the context of the program, the region ordering is needed to ensure that mux operations will only occur in "one direction." E.g., if two fresh flip values are created $x = \texttt{flip}^{\rho_1}()$ and $y = \texttt{flip}^{\rho_2}$, it is true that both $x \perp\!\!\!\perp y$ and $y \perp\!\!\!\perp x$. Thus it would seem reasonable that $\texttt{mux}(\texttt{cast}_S(x), y, \ldots)$ and $\texttt{mux}(\texttt{cast}_S(y), x, \ldots)$ should both be well typed. While they are both safe in isolation, the combination is problematic. Consider the results of each mux—they are both flip values, and they are both valid to reveal using $\texttt{cast}_P$ individually. However, the resulting values are correlated (revealing one tells you information about the distribution of the other), which violates the uniformity guarantee of all $\texttt{cast}_P$ results. By ordering the regions, we are essentially promising to only allow mux operations like this in one direction but not the other, and therefore uniformity is never violated for revealed flip values. For example, by requiring $\rho_1 \sqsubset \rho_2$ we allow the first mux above but not the second.

Type safety. $\lambda_{\textbf{obliv}}$ is type safe in the traditional sense, i.e., that a well-typed program will not get stuck. However, our interest is in the stronger property that type-safe $\lambda_{\textbf{obliv}}$ programs do not reveal secret information via inferences an adversary can draw from observing their execution. We state and prove this stronger property in the next section.

4 PROBABILISTIC MEMORY TRACE OBLIVIOUSNESS

Our main metatheoretic result of this paper is that $\lambda_{\textbf{obliv}}$'s type system ensures probabilistic memory trace obliviousness (PMTO). We begin by defining this property, and then in the remainder of the section carry out its proof.

4.1 What is PMTO?

Figure 7 presents a model of the adversary's view of a computation as a new class of values, expressions and traces that "hide" sub-expressions (written •) considered to be secret. We define obs for mapping ordinary expressions to adversary visible expressions, and likewise for values and traces. Secret bit expressions, secret bit values, and secret flip values all map to •. Compound values, expressions, stores, etc. call obs in recursive positions as expected.

$$\bullet v \in \bullet\text{value} ::= \ldots \mid \bullet \qquad \bullet\sigma \in \bullet\text{store} \triangleq \text{loc} \rightharpoonup \bullet\text{value} \qquad \bullet t \in \bullet\text{trace} ::= \epsilon \mid \bullet t \cdot \bullet\varsigma$$
$$\bullet e \in \bullet\text{exp} ::= \ldots \mid \bullet \qquad \bullet\varsigma \in \bullet\text{config} ::= \bullet\sigma, \bullet e$$

$$\text{obs} \in (\text{exp} \to \bullet\text{exp}) \times (\text{store} \to \bullet\text{store}) \times (\text{config} \to \bullet\text{config}) \times (\text{trace} \to \bullet\text{trace})$$

$$\begin{array}{ll}
\text{obs}(x) & \triangleq x \\
\text{obs}(\texttt{fun}_y(x:\tau).\ e) & \triangleq \texttt{fun}_y(x:\tau).\ \text{obs}(e) \\
\text{obs}(\texttt{bitv}_P(b)) & \triangleq \texttt{bitv}_P(b) \\
\text{obs}(\texttt{bitv}_S(b)) & \triangleq \bullet \\
\text{obs}(\texttt{flipv}(b)) & \triangleq \bullet \\
\text{obs}(\texttt{locv}(\iota)) & \triangleq \bullet \\
\text{obs}(b_P) & \triangleq b_P \\
\text{obs}(b_S) & \triangleq \bullet \\
\text{obs}(\texttt{flip}^\rho()) & \triangleq \texttt{flip}^\rho() \\
\text{obs}(\texttt{cast}_\ell(v)) & \triangleq \texttt{cast}_\ell(\text{obs}(v))
\end{array}$$

$$\begin{array}{ll}
\text{obs}(\texttt{mux}(e_1,e_2,e_3)) & \triangleq \texttt{mux}(\text{obs}(e_1),\text{obs}(e_2),\text{obs}(e_3)) \\
\text{obs}(\texttt{xor}(e_1,e_2)) & \triangleq \texttt{xor}(\text{obs}(e_1),\text{obs}(e_2)) \\
\text{obs}(\texttt{if}(e_1)\{e_2\}\{e_3\}) & \triangleq \texttt{if}(\text{obs}(e_1))\{\text{obs}(e_2)\}\{\text{obs}(e_3)\} \\
\text{obs}(\texttt{ref}(e)) & \triangleq \texttt{ref}(\text{obs}(e)) \\
\text{obs}(\texttt{read}(e)) & \triangleq \texttt{read}(\text{obs}(e)) \\
\text{obs}(\texttt{write}(e_1,e_2)) & \triangleq \texttt{write}(\text{obs}(e_1),\text{obs}(e_2)) \\
\text{obs}(\langle e_1,e_2\rangle) & \triangleq \langle \text{obs}(e_1),\text{obs}(e_2)\rangle \\
\text{obs}(\texttt{let } x = e_1 \texttt{ in } e_2) & \triangleq \texttt{let } x = \text{obs}(e_1) \texttt{ in } \text{obs}(e_2) \\
\text{obs}(\texttt{let } x,y = e_1 \texttt{ in } e_2) & \triangleq \texttt{let } x,y = \text{obs}(e_1) \texttt{ in } \text{obs}(e_2) \\
\text{obs}(e_1(e_2)) & \triangleq \text{obs}(e_1)(\text{obs}(e_2))
\end{array}$$

$$\text{obs}(\sigma) \triangleq \{\iota \mapsto \text{obs}(v) \mid \iota \mapsto v \in \sigma\} \qquad \text{obs}(\epsilon) \triangleq \epsilon$$
$$\text{obs}(\sigma, e) \triangleq \text{obs}(\sigma), \text{obs}(e) \qquad \text{obs}(t \cdot \varsigma) \triangleq \text{obs}(t) \cdot \text{obs}(\varsigma)$$

$$\widetilde{\text{obs}}(\tilde{t}) \triangleq \text{do } t \leftarrow \tilde{t}\ ; \text{return}(\text{obs}(t)) \qquad \widetilde{\text{obs}} \in \mathcal{D}(\text{trace}) \to \mathcal{D}(\bullet\text{trace})$$

Fig. 7. Adversary observability

Probabilistic memory trace obliviousness (PMTO), stated formally below, holds when observationally equivalent configurations induce distributions of traces that are themselves observationally equivalent after N steps, for any N.

Proposition 4.1 (Probabilistic Memory Trace Obliviousness (PMTO)).
If $\text{obs}(e_1) = \text{obs}(e_2)$, $\vdash e_1$ *and* $\vdash e_2$ *then:*
(1) $\text{nstep}_{\mathcal{D}}(N, \varnothing, e_1)$ *and* $\text{nstep}_{\mathcal{D}}(N, \varnothing, e_2)$ *are defined*
(2) $\widetilde{\text{obs}}(\text{nstep}_{\mathcal{D}}(N, \varnothing, e_1)) = \widetilde{\text{obs}}(\text{nstep}_{\mathcal{D}}(N, \varnothing, e_2))$

Typically noninterference properties are stated to account for an initial store, however, we only consider initial configurations which have an empty store, and where the expression contains no values (other than bound variables). The reason we do this is so that the initial configuration can be shared among both the standard semantics and the "mixed" semantics defined in the next section, which has a different representation for runtime values. This simplifies the formalism, and is just as strong as if PMTO was stated with an arbitrary starting store. The reason is that any store and expression with embedded location values can be re-encoded as a pure expression with no store:

$$\text{ENCODE}(\{\iota_1 \mapsto v_1, \ldots, \iota_n \mapsto v_n\}, e) \triangleq \begin{cases} \texttt{let } x_1 = \texttt{ref}(v_1) \texttt{ in} \\ \quad\vdots \\ \texttt{let } x_n = \texttt{ref}(v_n) \texttt{ in} \\ [x_1/\texttt{locv}(\iota_1), \ldots, x_n/\texttt{locv}(\iota_n)]e \end{cases}$$

Executing the encoded term is equivalent to executing the original term in the original store, and the encoding preserves equality of configurations modulo adversary observation due to secret bit literals and secret bit values both mapping to $\bullet$ in the definition of obs. Importantly, our *proof*

of PMTO considers intermediate configurations and non-empty stores, however these instances appear in strengthened induction hypotheses, are not restricted to source programs, and never need to be "shared" between standard and mixed semantics, which is the specific cause of the issue discussed here.

The remainder of this section works through our proof of PMTO. In a nutshell, the proof follows three steps. First, we define an alternative *mixed* semantics for $\lambda_{\textbf{obliv}}$ programs that simplifies inductive reasoning about the adversary's view. Second, we prove that key invariants about probabilistic values are ensured by typing (adapted to the mixed-semantics setting). Finally, we prove PMTO for the mixed semantics, and then show that it faithfully simulates the standard semantics and the adversary's view of its events.

4.2 Mixed Semantics

An intuitive approach to proving Proposition 4.1 is to prove that a single-step version of it holds for step, and then use that fact in an inductive proof over nstep. Unfortunately, proving the single-step version quickly runs into trouble. Consider a source program $\texttt{cast}_{\mathrm{P}}(\texttt{flip}())$ which steps to each of the expressions $\texttt{cast}_{\mathrm{P}}(\texttt{flip}_{\mathrm{v}}(\texttt{I}))$ and $\texttt{cast}_{\mathrm{P}}(\texttt{flip}_{\mathrm{v}}(\texttt{O}))$ with probability $1/2$. These expressions are low equivalent—the adversary's view of each is $\texttt{cast}_{\mathrm{P}}(\bullet)$. For single-step PMTO to be satisfied, each of these terms must step to an equivalent distribution. Unfortunately, they do not: The first produces a point distribution of the expression $\texttt{bit}_{\mathrm{P}}(\texttt{I})$ and the second produces a point distribution of the expression $\texttt{bit}_{\mathrm{P}}(\texttt{O})$, which are not observationally the same.

To address this problem, we define an alternative *mixed* semantics which embeds *distributional bit values* directly into (single) traces. Instead of the semantics of $\texttt{flip}^{\rho}()$ producing two possible outcomes, in the mixed semantics it produces just one: a single distributional value $\texttt{flip}_{\mathrm{v}}(\hat{b})$ where the $\hat{b}$ represents either $\texttt{I}$ or $\texttt{O}$ with equal probability. Doing this is like treating $\texttt{flip}^{\rho}()$ expressions lazily, until their contents are actually examined, and lines up (mixed) traces with the adversary's view $\bullet$.

The mixed semantics amends the syntax of $\texttt{flip}_{\mathrm{v}}$ and $\texttt{bit}_{\mathrm{v}\ell}$ values to be distributional (i.e., they contain $\hat{b}$ rather than just b). Other values from the standard semantics' syntax (top of Figure 5) are unchanged. As such, a distribution of pairs of bit values (say) is represented as pair of distributional bit values. To allow the values of the pair to be correlated, we represent them using what we call *intensional distributions*—an intensional distribution of type A is written $\mathcal{I}(A)$. We discuss these in depth in the next subsection.

The mixed semantics is shown in Figure 8. The mixed semantics step function $\underline{\text{step}}(N, \underline{\sigma}, \underline{e})$ maps a configuration, $\underline{\varsigma} \triangleq \langle \underline{\sigma}, \underline{e} \rangle$ to an intensional distribution of configurations $\mathcal{I}(\underline{\text{config}})$. The $\underline{\text{step}}$ takes an additional argument N which represents the number of steps of execution taken so far, and which factors into the way randomness is represented. Mixed semantics expressions (and values, etc.) are underlined to distinguish them from the standard semantics, and operations on distributional values are hatted.

Most of the cases for the mixed semantics are structurally the same as the standard semantics. The key differences are the handling of $\texttt{flip}^{\rho}()$ and $\texttt{cast}_{\ell}(\underline{v})$. For the first, the standard semantics samples from the fresh uniform distribution immediately, while the mixed semantics produces a single uniform distributional value. The definition of $\text{bit}(N)$ for intensional distributions uses N to produce a fresh intensional distribution of bits which only depends on the Nth random bit in "the universe." In essence, the universe of random bits is global and fixed, and hence two distributions which mention the Nth random bit for the same N are referring to the same source of randomness. This distributional value is sampled at the evaluation of $\texttt{cast}_{P}$, which matches the adversary's view of evaluation.

$$\boxed{\underline{\text{step}} \in \mathbb{N} \times \underline{\text{config}} \rightharpoonup \mathcal{I}(\underline{\text{config}})}$$

$$\begin{array}{ll}
\underline{\text{step}}(N, \underline{\sigma}, b_\ell) & \triangleq \text{return}(\underline{\sigma}, \texttt{bitv}_\ell(\text{return}(b))) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{flip}^\rho()) & \triangleq \text{return}(\underline{\sigma}, \texttt{flipv}(\text{bit}(N))) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{cast}_S(\texttt{flipv}(\hat{b}))) & \triangleq \text{return}(\underline{\sigma}, \texttt{bitv}_S(\hat{b})) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{cast}_P(\texttt{flipv}(\hat{b}))) & \triangleq \text{do } b \leftarrow \hat{b} \text{ ; return}(\underline{\sigma}, \texttt{bitv}_P(\text{return}(b))) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{mux}(\texttt{bitv}_{\ell_1}(\hat{b}_1), \texttt{bitv}_{\ell_2}(\hat{b}_2), \texttt{bitv}_{\ell_3}(\hat{b}_3))) & \triangleq \text{return}(\underline{\sigma}, \langle \texttt{bitv}_\ell(\widehat{\text{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)), \texttt{bitv}_\ell(\widehat{\text{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)) \rangle) \\
\quad \textit{where } \ell \triangleq \ell_1 \sqcup \ell_2 \sqcup \ell_3 & \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{mux}(\texttt{bitv}_\ell(\hat{b}_1), \texttt{flipv}(\hat{b}_2), \texttt{flipv}(\hat{b}_3))) & \triangleq \text{return}(\underline{\sigma}, \langle \texttt{flipv}(\widehat{\text{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)), \texttt{flipv}(\widehat{\text{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)) \rangle) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{xor}(\texttt{bitv}_{\ell_1}(\hat{b}_1), \texttt{flipv}(\hat{b}_2))) & \triangleq \text{return}(\underline{\sigma}, \texttt{flipv}(\hat{b}_1 \mathbin{\hat{\oplus}} \hat{b}_2)) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{if}(\texttt{bitv}_\ell(\hat{b}))\{\underline{e}_1\}\{\underline{e}_2\}) & \triangleq \text{do } b \leftarrow \hat{b} \text{ ; return}(\underline{\sigma}, \text{cond}(b, \underline{e}_1, \underline{e}_2)) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{ref}(\underline{v})) & \triangleq \text{return}(\underline{\sigma}[\iota \mapsto \underline{v}], \texttt{refv}(\iota)) \quad \textit{where } \iota \notin \text{dom}(\underline{\sigma}) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{read}(\texttt{refv}(\iota))) & \triangleq \text{return}(\underline{\sigma}, \underline{\sigma}(\iota)) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{write}(\texttt{refv}(\iota), \underline{v})) & \triangleq \text{return}(\underline{\sigma}[\iota \mapsto \underline{v}], \underline{\sigma}(\iota)) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{let } x = \underline{v} \texttt{ in } \underline{e}) & \triangleq \text{return}(\underline{\sigma}, \underline{e}[\underline{v}/x]) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{let } x_1, x_2 = \langle \underline{v}_1, \underline{v}_2 \rangle \texttt{ in } \underline{e}) & \triangleq \text{return}(\underline{\sigma}, \underline{e}[\underline{v}_1/x_1][\underline{v}_2/x_2]) \\
\underline{\text{step}}(N, \underline{\sigma}, (\underbrace{\texttt{fun}_y(x : \tau).\ \underline{e}}_{\underline{v}_1})(\underline{v}_2)) & \triangleq \text{return}(\underline{\sigma}, \underline{e}[\underline{v}_1/y][\underline{v}_2/x]) \\
\underline{\text{step}}(N, \underline{\sigma}, \underline{E}[\underline{e}]) & \triangleq \text{do } \underline{\sigma}', \underline{e}' \leftarrow \underline{\text{step}}(N, \underline{\sigma}, \underline{e}) \text{ ; return}(\underline{\sigma}', \underline{E}[\underline{e}']) \\
\underline{\text{step}}(N, \underline{\sigma}, \underline{v}) & \triangleq \text{return}(\underline{\sigma}, \underline{v})
\end{array}$$

$$\boxed{\underline{\text{nstep}} \in \mathbb{N} \times \underline{\text{config}} \rightharpoonup \mathcal{I}(\underline{\text{trace}})}$$

$$\begin{array}{ll}
\underline{\text{nstep}}(0, \underline{\varsigma}) & \triangleq \text{return}(\epsilon \cdot \underline{\varsigma}) \\
\underline{\text{nstep}}(N+1, \underline{\varsigma}) & \triangleq \text{do } \underline{t} \cdot \underline{\varsigma}' \leftarrow \underline{\text{nstep}}(N, \underline{\varsigma}) \text{ ; } \underline{\varsigma}'' \leftarrow \underline{\text{step}}(N+1, \underline{\varsigma}') \text{ ; return}(\underline{t} \cdot \underline{\varsigma}' \cdot \underline{\varsigma}'')
\end{array}$$

Fig. 8. Mixed Language Semantics, where $\hat{b} \in \mathcal{I}(\mathbb{B})$ is a distributional bit value (see text)

A secret literal will produce a point distribution on that literal. The semantic operations for `if`, `mux` and `xor` are lifted monadically to operate over distributions of secrets, e.g., $\hat{b}_1 \mathbin{\hat{\oplus}} \hat{b}_2 \triangleq \text{do } b_1 \leftarrow \hat{b}_1 \text{ ; } b_2 \leftarrow \hat{b}_2 \text{ ; return}(b_1 \oplus b_2)$. Other operations are as usual, e.g., let expressions and tuple elimination reduce via substitution and are not lifted to distributions.

4.3 Capturing Correlations with Intensional Distributions

At a high level, a distributional bit value $\hat{b}$ can be viewed as a lazy interpretation of a call $\texttt{flip}^\rho()$. To be sound, this interpretation must properly model conditional probabilities between variables.

Example. Consider the program $\texttt{let } x = \texttt{flip}^\rho() \texttt{ in } \langle \texttt{cast}_P(x), \texttt{cast}_P(x) \rangle$.[2] After two evaluation steps in the standard semantics, the program will be reduced to either $\langle \texttt{cast}_P(\texttt{flipv}(\texttt{I})), \texttt{cast}_P(\texttt{flipv}(\texttt{I})) \rangle$ or $\langle \texttt{cast}_P(\texttt{flipv}(\texttt{O})), \texttt{cast}_P(\texttt{flipv}(\texttt{O})) \rangle$, with equal probability. The standard rules for $\texttt{cast}_P$ would then yield (equally likely) $\langle \texttt{bitv}_P(\texttt{I}), \texttt{bitv}_P(\texttt{I}) \rangle$ and $\langle \texttt{bitv}_P(\texttt{O}), \texttt{bitv}_P(\texttt{O}) \rangle$. In the mixed semantics this program will evaluate in two steps to $\langle \texttt{cast}_P(\texttt{flipv}(\hat{b})), \texttt{cast}_P(\texttt{flipv}(\hat{b})) \rangle$ where $\hat{b}$ is a distributional value. At this point, the mixed semantics rule for $\texttt{cast}_P$ uses monadic bind to sample $\hat{b}$ to yield some b (which is either `I` or `O`) and return it as a point distribution. The semantics needs to "remember" the bit

[2] Although this program violates affinity and would be rejected for that reason by our type system, its runtime semantics is well-defined and serves as a helpful demonstration.

$$
\begin{array}{l}
a \in \quad A \\
\hat{x} \in \mathcal{I}(A) ::= a \mid \langle \hat{x}\ \hat{x} \rangle \\
p \in \mathrm{rpath} ::= \cdot \mid \textcircled{H} :: p \mid \textcircled{T} :: p
\end{array}
$$

$$
\begin{array}{ll}
\_[\_] & \in \mathcal{I}(A) \times \mathrm{rpath} \rightharpoonup A \\
a[p] & \triangleq a \\
\langle \hat{x}_1\ \hat{x}_2 \rangle[\textcircled{H} :: p] & \triangleq \hat{x}_1[p] \\
\langle \hat{x}_1\ \hat{x}_2 \rangle[\textcircled{T} :: p] & \triangleq \hat{x}_2[p]
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{support} & \in \mathcal{I}(A) \to \wp(A) \\
\mathrm{support}(\hat{x}) & \triangleq \{a \mid \hat{x}[p] = a\}
\end{array}
$$

$$
\begin{array}{ll}
\pi_1 & \in \mathcal{I}(A) \to \mathcal{I}(A) \\
\pi_1(a) & \triangleq a \\
\pi_1(\langle \hat{x}_1\ \hat{x}_2 \rangle) & \triangleq \hat{x}_1
\end{array}
$$

$$
\begin{array}{ll}
\pi_2 & \in \mathcal{I}(A) \to \mathcal{I}(A) \\
\pi_2(a) & \triangleq a \\
\pi_2(\langle \hat{x}_1\ \hat{x}_2 \rangle) & \triangleq \hat{x}_2
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{height} & \in \mathcal{I}(A) \to \mathbb{N} \\
\mathrm{height}(a) & \triangleq 0 \\
\mathrm{height}(\langle \hat{x}_1\ \hat{x}_2 \rangle) & \triangleq 1 + \max(\mathrm{height}(\hat{x}_1), \mathrm{height}(\hat{x}_2))
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{length} & \in \mathrm{rpath} \to \mathbb{B} \\
\mathrm{length}(\cdot) & \triangleq 0 \\
\mathrm{length}(\_ :: p) & \triangleq 1 + \mathrm{length}(p)
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{bit} & \in \mathbb{N} \to \mathcal{I}(\mathbb{B}) \\
\mathrm{bit}(0) & \triangleq \langle \mathtt{I}\ \mathtt{O} \rangle \\
\mathrm{bit}(N+1) & \triangleq \langle \mathrm{bit}(N)\ \mathrm{bit}(N) \rangle
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{return} & \in A \to \mathcal{I}(A) \\
\mathrm{return}(a) & \triangleq a
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{bind} & \in \mathcal{I}(A) \times (A \to \mathcal{I}(B)) \to \mathcal{I}(B) \\
\mathrm{bind}(a, f) & \triangleq f(a) \\
\mathrm{bind}(\langle \hat{x}_1\ \hat{x}_2 \rangle, f) & \triangleq \langle \mathrm{bind}(\hat{x}_1, \pi_1 \circ f)\ \mathrm{bind}(\hat{x}_2, \pi_2 \circ f) \rangle
\end{array}
$$

$$
\begin{array}{ll}
\Pr\left[\overline{\hat{x} \doteq x}\right] & \triangleq \dfrac{\left|\{p \mid \mathrm{length}(p) = h, \overline{\hat{x}[p] = x}\}\right|}{2^h} \quad \textit{where} \quad h \triangleq \max(\overline{\mathrm{height}(\hat{x})}) \\
\Pr\left[\overline{\hat{x} \doteq x} \,\middle|\, \overline{\hat{y} \doteq y}\right] & \triangleq \dfrac{\Pr\left[\overline{\hat{x} \doteq x}, \overline{\hat{y} \doteq y}\right]}{\Pr\left[\overline{\hat{y} \doteq y}\right]}
\end{array}
$$

Fig. 9. Intensional Distributions

chosen for the first `castP` so that when it samples the second, the same bit is returned. Sampling independently would yield incorrect outcomes such as $\langle \mathtt{bitvP}(\mathtt{O}), \mathtt{bitvP}(\mathtt{I}) \rangle$.

Intensional distributions. We avoid this problem by being eager in the choice of *which* coin flip a distributional value represents, and only lazy in the sampling of values from it. Doing so is made possible by what we call *intensional distributions*. As shown in the upper left of Figure 9, an intensional distribution $\mathcal{I}(A)$ over a set A is a binary tree with elements of A at the leaves. It represents a distribution as a function from input entropy—a sequence of coin flips—to a result in A. Each node $\langle \hat{x}_1\ \hat{x}_2 \rangle$ in the tree represents two sets of worlds determined by the result of a coin flip: the left side $\hat{x}_1$ defines the worlds in which the coin was heads, and the right side $\hat{x}_2$ defines those in which it was tails. Each level of the tree represents a distinct coin flip, with the earliest coin flip at the root, and later coin flips at lower levels. The height of a tree represents an upper bound on the number of coin flips upon which a distribution's values depends. Each path through the tree is a possible world.

For example, $\langle\langle 3\ 4 \rangle\ \langle 3\ 5 \rangle\rangle$ is a distribution of numbers in an execution in which two coins have been flipped. There are four possible worlds. The $\langle 3\ 4 \rangle$ defines the worlds in which the 0th coin came up heads. Thus the 3 is the outcome in the world when both coins came up heads, while the 4 is the outcome in which the 0th coin was heads but the 1th coin was tails. The $\langle 3\ 5 \rangle$ defines the worlds in which the 0th coin came up tails, with 3 the outcome when the 1th coin is heads, and 5 when it's tails.

We can derive the probabilities of these outcomes by counting the number of paths that result in the same outcome. In the example, 3 has probability $\frac{1}{2}$; while 4 has probability $\frac{1}{4}$; and 5 has

probability $\frac{1}{4}$. Importantly, intensional distributions have enough structure to represent correlations: We can see that we always get a 3 when the 1th coin flip is heads, regardless of whether the 0th coin flip was heads or tails. Conversely, the distribution ‹‹3 3› ‹4 5›› ascribes outcomes 3, 4, and 5 the same probabilities as ‹‹3 4› ‹3 5››, but represents the situation in which the we always get 3 when 0th coin flip is heads, regardless of the 1th flip's outcome. An equivalent representation of ‹‹3 3› ‹4 5›› is ‹3 ‹4 5››. Although in the latter representation 3 only appears once, it is logically extended to the larger sub-tree ‹3 3› for the purposes of counting and probability ratios. To compute a probability, all paths are considered of a fixed length equal to the height of the tree, and shorter sub-trees are extended in this way with copies of leaves which appear at shorter height.

In the figure, a path p through the tree is defined as a sequence of coin flip outcomes, either Ⓗ or Ⓣ. The operation $\hat{x}[p]$ follows a path p through the tree $\hat{x}$ going left on Ⓗ and right on Ⓣ. When a leaf a is reached, it is simply returned, per the case $a[p]$; if p happens to not be $\cdot$, returning a is tantamount to extending the tree logically, as mentioned above. Computing the probability of an outcome x for an intensional distribution $\hat{x}$ is shown at the bottom of the figure. As with the example above, it counts up the number of paths that have outcome x, scaled them by the total possible worlds. The probability of an event involving multiple intensional distributions is similar. Conditional probability works as usual.

Finally, looking at the middle right of the figure, consider the monadic operations used by the semantics in Figure 8. The $\mathrm{bit}(N)$ operation produces a uniform distribution of bits following the Nth coin flip, where the outcomes are entirely determined by the Nth flip, i.e., independent of the flips that preceded it, which appear higher in the tree. $\mathrm{return}(a)$ simply returns a—this corresponds to a point distribution of a since it is the outcome in all possible worlds (recall $a[p] = a$ for all p). Lastly, $\mathrm{bind}(\hat{x}, f)$ applies f to each possible world in $\hat{x}$, gathering up the results in an intensional distribution tree that is of equal or greater height to that of $\hat{x}$; the height could grow if f returns a tree larger than $\hat{x}$. If $\hat{x}$ is height h_1 and $f(x)$ is height h_2 or less for all x, then $\mathrm{bind}(\hat{x}, f)$ is height $\max(h_1, h_2)$ and $\mathrm{bind}(\hat{x}, f)[p] = f(\hat{x}[p])[p]$.

Example revisited. Reconsider the example $\texttt{let}\ x = \texttt{flip}^{\rho}()\ \texttt{in}\ \langle \texttt{cast}_{\mathrm{P}}(x), \texttt{cast}_{\mathrm{P}}(x)\rangle$. According to the mixed semantics starting with $N = 0$, $\texttt{flip}^{\rho}()$ evaluates to $\texttt{flipv}(\texttt{‹I 0›})$, which is then (as precipitated by nstep) substituted for x in the body of the let, producing $\langle \texttt{cast}_{\mathrm{P}}(\texttt{flipv}(\texttt{‹I 0›})), \texttt{cast}_{\mathrm{P}}(\texttt{flipv}(\texttt{‹I 0›}))\rangle$. Now we apply the context rule for $\underline{E}[\underline{e}]$ where $\underline{E}$ is $\langle [], \texttt{cast}_{\mathrm{P}}(\texttt{flipv}(\texttt{‹I 0›}))\rangle$ and $\underline{e}$ is $\texttt{cast}_{\mathrm{P}}(\texttt{flipv}(\texttt{‹I 0›}))$. The rule invokes step on the latter, which performs $\mathrm{do}\ b \leftarrow \texttt{‹I 0›}\ ;\ \mathrm{return}(\underline{\sigma}, \texttt{bitv}_{\mathrm{P}}(\mathrm{return}(b)))$ per the rule for $\texttt{cast}_{\mathrm{P}}$. Per the definitions of bind and return, this will return the intensional *distribution of configurations* $\texttt{‹}(\underline{\sigma}, \texttt{bitv}_{\mathrm{P}}(\texttt{I}))\ (\underline{\sigma}, \texttt{bitv}_{\mathrm{P}}(\texttt{0}))\texttt{›}$. Back to the context rule, its use of bind will re-package up these possibilities with $\underline{E}$:

$$\texttt{‹}(\underline{\sigma}, \langle \texttt{bitv}_{\mathrm{P}}(\texttt{I}), \texttt{cast}_{\mathrm{P}}(\texttt{flipv}(\texttt{‹I 0›}))\rangle)\ (\underline{\sigma}, \langle \texttt{bitv}_{\mathrm{P}}(\texttt{0}), \texttt{cast}_{\mathrm{P}}(\texttt{flipv}(\texttt{‹I 0›}))\rangle)\texttt{›}$$

In this distribution of configurations, there are two worlds—the configuration on the left occurs when the 0th coin flip is heads, and the one on the right when it is tails. Inside of each of these configurations is a distributional value $\texttt{flipv}(\texttt{‹I 0›})$, where once again the left side is due to the coin flip being heads, and the right side being tails. *Both are relative to the same coin flip.* As such, there are two "dead" paths in the inner trees: the right-branch of the left distributional value, and the left branch of the right distributional value, shown here with bullets:

$$\texttt{‹}(\underline{\sigma}, \langle \texttt{bitv}_{\mathrm{P}}(\texttt{I}), \texttt{cast}_{\mathrm{P}}(\texttt{flipv}(\texttt{‹I •›}))\rangle)\ (\underline{\sigma}, \langle \texttt{bitv}_{\mathrm{P}}(\texttt{0}), \texttt{cast}_{\mathrm{P}}(\texttt{flipv}(\texttt{‹• 0›}))\rangle)\texttt{›}$$

The next step of the computation will force the distributional value to be I in the left branch and 0 in the right branch. Here's how. First, the definition of nstep is a bind on the above distribution of configurations with step as the function f passed to bind. The definition of bind constructs a new distribution tree which calls step on the left configuration, and then takes the left branch (π_1) of

$$\boxed{\lceil \_ \rceil \in (\underline{\text{exp}} \to \mathcal{I}(\text{exp})) \times (\underline{\text{store}} \to \mathcal{I}(\text{store})) \times (\underline{\text{config}} \to \mathcal{I}(\text{config})) \times (\underline{\text{trace}} \to \mathcal{I}(\text{trace}))}$$

$$\begin{array}{ll}
\lceil x \rceil & \triangleq \text{return}(x) \\
\lceil \texttt{fun}_y(x : \tau).\ \underline{e} \rceil & \triangleq \text{do } e \leftarrow \lceil \underline{e} \rceil \text{ ; return}(\texttt{fun}_y(x : \tau).\ e) \\
\lceil \texttt{bitv}_\ell(\hat{b}) \rceil & \triangleq \text{do } b \leftarrow \hat{b} \text{ ; return}(\texttt{bitv}_\ell(b)) \\
\lceil \texttt{flipv}(\hat{b}) \rceil & \triangleq \text{do } b \leftarrow \hat{b} \text{ ; return}(\texttt{flipv}(b)) \\
\lceil \texttt{locv}(\iota) \rceil & \triangleq \text{return}(\texttt{locv}(\iota)) \\
\lceil b_\ell \rceil & \triangleq \text{return}(b_\ell) \\
\lceil \texttt{flip}^\rho() \rceil & \triangleq \text{return}(\texttt{flip}^\rho()) \\
\lceil \texttt{cast}_\ell(\underline{v}) \rceil & \triangleq \text{do } v \leftarrow \lceil \underline{v} \rceil \text{ ; return}(\texttt{cast}_\ell(v)) \\
\lceil \texttt{mux}(\underline{e}_1, \underline{e}_2, \underline{e}_3) \rceil & \triangleq \text{do } e_1 \leftarrow \lceil \underline{e}_1 \rceil \text{ ; } e_2 \leftarrow \lceil \underline{e}_2 \rceil \text{ ; } e_3 \leftarrow \lceil \underline{e}_3 \rceil \text{ ; return}(\texttt{mux}(e_1, e_2, e_3)) \\
\lceil \texttt{xor}(\underline{e}_1, \underline{e}_2) \rceil & \triangleq \text{do } e_1 \leftarrow \lceil \underline{e}_1 \rceil \text{ ; } e_2 \leftarrow \lceil \underline{e}_2 \rceil \text{ ; return}(\texttt{xor}(e_1, e_2)) \\
\lceil \texttt{if}(\underline{e}_1)\{\underline{e}_2\}\{\underline{e}_3\} \rceil & \triangleq \text{do } e_1 \leftarrow \lceil \underline{e}_1 \rceil \text{ ; } e_2 \leftarrow \lceil \underline{e}_2 \rceil \text{ ; } e_3 \leftarrow \lceil \underline{e}_3 \rceil \text{ ; return}(\texttt{if}(e_1)\{e_2\}\{e_3\}) \\
\lceil \texttt{ref}(\underline{e}_1) \rceil & \triangleq \text{do } e_1 \leftarrow \lceil \underline{e}_1 \rceil \text{ ; return}(\texttt{ref}(e_1)) \\
\lceil \texttt{read}(\underline{e}_1) \rceil & \triangleq \text{do } e_1 \leftarrow \lceil \underline{e}_1 \rceil \text{ ; return}(\texttt{read}(e_1)) \\
\lceil \texttt{write}(\underline{e}_1, \underline{e}_2) \rceil & \triangleq \text{do } e_1 \leftarrow \lceil \underline{e}_1 \rceil \text{ ; } e_2 \leftarrow \lceil \underline{e}_2 \rceil \text{ ; return}(\texttt{write}(e_1, e_2)) \\
\lceil \langle \underline{e}_1, \underline{e}_2 \rangle \rceil & \triangleq \text{do } e_1 \leftarrow \lceil \underline{e}_1 \rceil \text{ ; } e_2 \leftarrow \lceil \underline{e}_2 \rceil \text{ ; return}(\langle e_1, e_2 \rangle) \\
\lceil \texttt{let } x = \underline{e}_1 \texttt{ in } \underline{e}_2 \rceil & \triangleq \text{do } e_1 \leftarrow \lceil \underline{e}_1 \rceil \text{ ; } e_2 \leftarrow \lceil \underline{e}_2 \rceil \text{ ; return}(\texttt{let } x = e_1 \texttt{ in } e_2) \\
\lceil \texttt{let } x, y = \underline{e}_1 \texttt{ in } \underline{e}_2 \rceil & \triangleq \text{do } e_1 \leftarrow \lceil \underline{e}_1 \rceil \text{ ; } e_2 \leftarrow \lceil \underline{e}_2 \rceil \text{ ; return}(\texttt{let } x, y = e_1 \texttt{ in } e_2) \\
\lceil \underline{e}_1(\underline{e}_2) \rceil & \triangleq \text{do } e_1 \leftarrow \lceil \underline{e}_1 \rceil \text{ ; } e_2 \leftarrow \lceil \underline{e}_2 \rceil \text{ ; return}(e_1(e_2))
\end{array}$$

$$\lceil \varnothing \rceil \triangleq \text{return}(\varnothing) \qquad \lceil \{\iota \mapsto \underline{v}\} \uplus \underline{\sigma} \rceil \triangleq \text{do } v \leftarrow \lceil \underline{v} \rceil \text{ ; } \sigma \leftarrow \lceil \underline{\sigma} \rceil \text{ ; return}(\{\iota \mapsto v\} \uplus \sigma)$$

$$\lceil \underline{\sigma}, \underline{e} \rceil \triangleq \text{do } \sigma \leftarrow \underline{\sigma} \text{ ; } e \leftarrow \underline{e} \text{ ; return}(\sigma, e) \qquad \lceil \underline{t} \cdot \underline{\varsigma} \rceil \triangleq \text{do } t \leftarrow \underline{t} \text{ ; } \varsigma \leftarrow \underline{\varsigma} \text{ ; return}(t \cdot \varsigma)$$

$$\lceil \epsilon \rceil \triangleq \text{return}(\epsilon) \qquad \hat{\lceil} \hat{\underline{t}} \hat{\rceil} \triangleq \text{do } \underline{t} \leftarrow \hat{\underline{t}} \text{ ; } \lceil \underline{t} \rceil \qquad \boxed{\hat{\lceil} \_ \hat{\rceil} \in \mathcal{I}(\underline{\text{trace}}) \to \mathcal{I}(\text{trace})}$$

Fig. 10. Mixed Semantics Projection

the tree that comes back, and likewise for the right configuration and the right branch that comes back (π_2). Here step will invoke cast and context rules similarly as before, returning a two-element tree with $\texttt{bitv}_P(\texttt{I})$ on the left and $\texttt{bitv}_P(\texttt{0})$ on the right. These occurrences of π_1 and π_2 "pick" the left (I case) and right (0 case), respectively, resulting in the final configuration:

$$\langle(\underline{\sigma}, \langle \texttt{bitv}_P(\texttt{I}), \texttt{bitv}_P(\texttt{I})\rangle)\ (\underline{\sigma}, \langle \texttt{bitv}_P(\texttt{0}), \texttt{bitv}_P(\texttt{0})\rangle)\rangle$$

Simulation. The concept of "unreachable" paths in a distributional value is captured by a projection operation which "flattens" a distribution of mixed terms (which have distributional values) into a distribution of standard terms (which do not have distributional values). This projection will (1) discard unreachable paths of distributional values, and (2) corresponds to evaluation in the standard semantics instantiated with the intensional distribution monad, which we prove as a simulation theorem.

Projection is defined in Figure 10. The definition is a straightforward use of bind to recursively flatten embedded distributional values. In our example, the projection of the mixed term before the step shows what is left after discarding the unreachable distribution elements:

$$\begin{array}{l}
\hat{\lceil} \langle(\underline{\sigma}, \langle \texttt{bitv}_P(\texttt{I}), \texttt{castp}(\texttt{flipv}(\langle \texttt{I } \texttt{0} \rangle))\rangle)\ (\underline{\sigma}, \langle \texttt{bitv}_P(\texttt{0}), \texttt{castp}(\texttt{flipv}(\langle \texttt{I } \texttt{0} \rangle))\rangle)\rangle \hat{\rceil} \\
\quad = \langle(\underline{\sigma}, \langle \texttt{bitv}_P(\texttt{I}), \texttt{castp}(\texttt{flipv}(\texttt{I}))\rangle)\ (\underline{\sigma}, \langle \texttt{bitv}_P(\texttt{0}), \texttt{castp}(\texttt{flipv}(\texttt{0}))\rangle)\rangle
\end{array}$$

and where the RHS corresponds exactly to the step of computation using the standard semantics.

The connection between standard and mixed semantics via projection is captured in a simulation theorem. The following is specialized to source programs, but we have proved the general case for all mixed terms.

Lemma 4.2 (Intensional Simulation). *If e is a source expression, then* $\lceil \underline{\text{nstep}}(N, \varnothing, e) \rceil = \text{nstep}_{\mathcal{I}}(N, \varnothing, e)$.

The proof is by induction over the monadic structure of nstep. The induction appeals to a single-step version of simulation (i.e., over step) over mixed term $\underline{e}$, which is proved by a simple induction on $\underline{e}$, appealing to monad laws and the definition of projection.

The simulation lemma establishes simulation of the mixed semantics w.r.t. the standard semantics using an intensional distribution monad. In order to relate to the "ground truth", which uses the denotational probability monad $\mathcal{D}$, we prove another lemma.

Lemma 4.3 (Ground Truth Simulation). $\Pr\left[\text{nstep}_{\mathcal{I}}(N, \varnothing, e) \doteq \varsigma\right] = \Pr\left[\text{nstep}_{\mathcal{D}}(N, \varnothing, e) \doteq \varsigma\right]$.

4.4 Mixed Semantics Typing

The $\lambda_{\textbf{obliv}}$ type system aims to ensure that a `cast`$_P$ can always produce `I` and `O` with equal likelihood, meaning neither outcome leaks any information. We establish this invariant in the PMTO proof as a consequence of type preservation of mixed terms. The mixed term typing judgment essentially extends typing of source-program expressions (Figure 6), but adds some additional elements and considers non-source values.

The judgment has the form $\Psi, \Phi, \Sigma \vdash \underline{\varsigma} : \tau, \Psi$, and is shown at the bottom of Figure 11. Here, Σ is a *store context*, which is a map from store locations to types; it is used to type the store $\underline{\sigma}$ in rules Store-Cons and LocV as usual. Φ represents *public knowledge*, and it encodes the sequence of particular evaluation steps taken to reach the present one. The type system reasons about the probability of outcomes of distributional values conditioned on (adversary visible) public knowledge. The Ψ is an *fbset*, which is a technical device used to collect all the distributional bit values $\hat{b}$ that appear in $\underline{\varsigma}$. Per the top of the figure, the fbset is a pair (Ψ^F, Ψ^B), where Ψ^F is a *flipset* containing those $\hat{b}$ that appear inside of flip values, and Ψ^B is a *bitset* containing those $\hat{b}$ inside bit values. The latter is organized as a map from a region ρ to a set of bit values in that region. The Ψ to the right of the turnstile contains all of the flip and secret bit values in the configuration itself, while the Ψ to the left of it captures those in the context.

The expression typing judgment $\Psi, \Phi, \Sigma, \Gamma \vdash \underline{e} : \tau \,;\, \Gamma, \Psi$ is similar but includes variable contexts Γ as in the source-program rules. We can see secret bit values being added to Ψ^B in the BitV-S rule, where Ψ^B is the singleton map from ρ, the region of the bit value, to $\{\hat{b}\}$, while Ψ^F is empty. Conversely, in the FlipV rule Ψ^B is empty while Ψ^F is the singleton set $\{\hat{b}\}$. We can see the maintenance of Ψ to the left of the turnstile in the Tup rule. Recursively typing the pair's left component $\underline{e}_1$ yields fbset Ψ_1 to the right of the turnstile, which is used when typing $\underline{e}_2$, and vice versa; the Store-Cons rule has a similar mutual recursion between the store and the expression. These rules combine two fbsets using the $\uplus$ operator, defined at the top of the figure. It acts as disjoint union for flipsets but normal union for bitsets; this mirrors the use of affinity to ensure flips are not duplicated.

The key invariants ensured by typing are defined by the judgment $\Psi, \Phi \vdash \hat{b} : \texttt{flip}^{\rho}$, which is invoked by expression-typing rule FlipV and defined in the Flip-Value rule. This judgment establishes that in a configuration reached by an execution path Φ the flip value $\hat{b}$ is uniformly distributed (first premise), and that it can be typed at region ρ because it is properly independent of the other secret bit values in smaller regions $\Psi^B(\{\rho' \mid \rho' \sqsubset \rho\})$ and flip values Ψ^F (second premise). Conditional

$$\Psi^F \in \text{flipset} \triangleq \wp(\mathcal{I}(\mathbb{B})) \quad \Psi^B \in \text{bitset} \triangleq \mathbb{R} \to \wp(\mathcal{I}(\mathbb{B})) \quad \Psi \in \text{fbset} ::= \Psi^F, \Psi^B \quad \Phi \in \text{pub-kn} ::= \overline{\hat{\varsigma} \doteq \underline{\varsigma}}$$

$$(\Psi_1^F, \Psi_1^B) \uplus (\Psi_2^F, \Psi_2^B) \triangleq (\Psi_1^F \uplus \Psi_2^F), (\Psi_1^B \cup \Psi_2^B)$$

$$\left[\overline{\hat{x}} \perp\!\!\!\perp \overline{\hat{y}} \;\middle|\; \overline{\hat{z} \doteq z}\right] \overset{\triangle}{\iff} \forall \overline{x}, \overline{y}.\ \Pr\left[\overline{\hat{x} \doteq x}, \overline{\hat{y} \doteq y} \;\middle|\; \overline{\hat{z} \doteq z}\right] = \Pr\left[\overline{\hat{x} \doteq x} \;\middle|\; \overline{\hat{z} \doteq z}\right] \Pr\left[\overline{\hat{y} \doteq y} \;\middle|\; \overline{\hat{z} \doteq z}\right]$$

Flip-Value

$$\frac{\Pr\left[\hat{b} \doteq \mathtt{I} \;\middle|\; \Phi\right] = 1/2 \qquad \left[\hat{b} \perp\!\!\!\perp \Psi^F, \Psi^B(\{\rho' \mid \rho' \sqsubset \rho\})\right]}{\Psi^F, \Psi^B, \Phi \vdash \hat{b} : \mathtt{flip}^\rho} \qquad \boxed{\Psi, \Phi \vdash \hat{b} : \mathtt{flip}^\rho}$$

BitV-P

$$\frac{}{\Psi, \Phi, \Sigma, \Gamma \vdash \mathtt{bitv}_P(\mathrm{return}(b)) : \mathtt{bit}_P^\perp \;;\; \Gamma, \varnothing, \varnothing}$$

BitV-S

$$\frac{}{\Psi, \Phi, \Sigma, \Gamma \vdash \mathtt{bitv}_S(\hat{b}) : \mathtt{bit}_S^\rho \;;\; \Gamma, \varnothing, \{\rho \mapsto \{\hat{b}\}\}}$$

FlipV

$$\frac{\Psi, \Phi \vdash \hat{b} : \mathtt{flip}^\rho}{\Psi, \Phi, \Sigma, \Gamma \vdash \mathtt{flipv}(\hat{b}) : \mathtt{flip}^\rho \;;\; \Gamma, \{\hat{b}\}, \varnothing}$$

LocV

$$\frac{\Sigma(\iota) = \tau}{\Psi, \Phi, \Sigma, \Gamma \vdash \mathtt{locv}(\iota) : \tau \;;\; \Gamma, \varnothing, \varnothing}$$

Tup

$$\cdots \qquad \frac{\Psi \uplus \Psi_2, \Phi, \Sigma, \Gamma \vdash \underline{e}_1 : \tau_1 \;;\; \Gamma', \Psi_1 \qquad \Psi \uplus \Psi_1, \Phi, \Sigma, \Gamma' \vdash \underline{e}_2 : \tau_2 \;;\; \Gamma'', \Psi_2}{\Psi, \Phi, \Sigma, \Gamma \vdash \langle \underline{e}_1, \underline{e}_2 \rangle : \tau_1 \times \tau_2 \;;\; \Gamma'', \Psi_1 \uplus \Psi_2} \qquad \cdots$$

Store-Empty

$$\frac{}{\Psi, \Phi, \Sigma \vdash \varnothing \;;\; \varnothing, \varnothing}$$

Store-Cons

$$\frac{\Psi \uplus \Psi_\sigma, \Phi, \Sigma, \varnothing \vdash \underline{v} : \Sigma(\iota) \;;\; \varnothing, \Psi_v \qquad \Psi \uplus \Psi_v, \Phi, \Sigma, \varnothing \vdash \underline{\sigma} \;;\; \Psi_\sigma}{\Psi, \Phi, \Sigma \vdash \{\iota \mapsto \underline{v}\} \uplus \underline{\sigma} \;;\; \Psi_v \uplus \Psi_\sigma} \qquad \boxed{\Psi, \Phi, \Sigma \vdash \underline{\sigma} \;;\; \Psi}$$

Config

$$\frac{\Psi \uplus \Psi_e, \Phi, \Sigma \vdash \underline{\sigma} \;;\; \Psi_\sigma \qquad \Psi \uplus \Psi_\sigma, \Phi, \Sigma, \varnothing \vdash \underline{e} : \tau \;;\; \varnothing, \Psi_e}{\Psi, \Phi, \Sigma \vdash \underline{\sigma}, \underline{e} : \tau \;;\; \Psi_\sigma \uplus \Psi_e} \qquad \boxed{\Phi, \Sigma \vdash \underline{\varsigma} : \tau, \Psi}$$

Fig. 11. Mixed Semantics Typing

independence is defined in the figure in the usual way—the overbar notation represents some sequence of random variables and/or condition events.

We prove a type preservation lemma to establish that these (and other) invariants are preserved.

Lemma 4.4 (Type Preservation). *If* $\Phi, \Sigma \vdash \underline{\varsigma} : \tau, \Psi$ *and* $\underline{\varsigma}' \in \mathrm{support}(\mathrm{step}_{\mathcal{I}}(N, \underline{\varsigma}))$ *then there exists* Σ' *and* Ψ' *s.t.* $\Phi', \Sigma' \vdash \underline{\varsigma}' : \tau, \Psi'$ *where* $\Phi' \triangleq \left[\Phi, \mathrm{step}_{\mathcal{I}}(N, \underline{\varsigma}) \doteq \underline{\varsigma}'\right]$.

When a configuration takes a step, the resulting configuration is well-typed under a new, updated public knowledge Φ'. This new Φ' is not arbitrary—it is fixed to be the old public knowledge plus a new condition that encodes that the current transition being considered $\underline{\varsigma}'$ occurred. This new public knowledge appears in exactly this form as a proof obligation in a later lemma. The new Σ' and Ψ' are new store typings (in case new references were allocated), and the new fbset (in case flip values were either created or consumed).

The key property established by type preservation is that flip values remain well-typed. Recall that the first premise of Flip-Value—uniformity—is crucial in establishing that it is safe to reveal the flip via the $\texttt{cast}_P$ coercion to a public bit. The second premise is crucial in re-establishing the first premise after some *other* flip has been revealed. When another flip is revealed, this information will be added to public knowledge, and it is not true that uniformity conditioned on the current public knowledge Φ implies uniformity in some new Φ' after new knowledge has been learned. Because the second premise establishes independence from all other flips, we are able restablish the first premise via the second after some other flip is revealed.

4.5 Proving PMTO

To prove PMTO (Proposition 4.1) we first prove a variant of it for the mixed semantics, and then apply a few more lemmas to show that PMTO holds for the standard semantics too.

Lemma 4.5 (PMTO (Mixed)). *If e_1 and e_2 are source expressions, $e_1 \sim e_2$, $\vdash e_1$ and $\vdash e_2$, then $\underline{\text{nstep}}(N, \varnothing, e_1) \approx_\sim \underline{\text{nstep}}(N, \varnothing, e_2)$.*

The judgment $\underline{e_1} \sim \underline{e_2}$ in the premise indicates that the two expressions are *low equivalent*, meaning that the adversary cannot tell them apart. The definition of this judgment is basically standard (given in the Appendix) and we can easily prove that it is implied by $\text{obs}(e_1) = \text{obs}(e_2)$ for source expressions (i.e., no runtime values). Mixed PMTO is establishes equivalence of the distributions of mixed configurations modulo low-equivalence. We define two distributions as equivalent modulo an underlying equivalence relation as follows:

$$\hat{x}_1 \approx_{\sim_A} \hat{x}_2 \stackrel{\Delta}{\iff} \forall x. \left(\sum_{x' | x' \sim_A x} \Pr[\hat{x}_1 \doteq x']\right) = \left(\sum_{x' | x' \sim_A x} \Pr[\hat{x}_2 \doteq x']\right)$$

This definition captures the idea that two distributions are equivalent when, for any equivalence class within the relation (represented by element x), each distribution assigns equal mass to the whole class. For Mixed PMTO, the relation $\sim_A$ is instantiated to low equivalence, which we write just as $\sim$. When the underlying relation is equality, we recover the usual notion of distribution equivalence: equality of probability mass functions.

We prove Mixed PMTO by induction over steps N and then unfolding the monadic definition of $\underline{\text{nstep}}(N+1)$. The induction appeals to a single-step PMTO proof. (As mentioned in Section 4.2, such a proof would not have been possible in the standard semantics.) This proof expects well-typed configurations, which we get from Type Preservation.

A final major lemma in our PMTO proof is a notion of soundness for low-equivalence on mixed terms, that is, equivalence modulo $\sim$ for distributions of mixed traces implies equality of adversary-observable traces in the standard semantics:

Lemma 4.6 (Low-equivalence Soundness). *If $\underline{\hat{t}_1} \approx_\sim \underline{\hat{t}_2}$ then $\widehat{\text{obs}}(\lceil \underline{\hat{t}_1} \rceil) \approx_= \widehat{\text{obs}}(\lceil \underline{\hat{t}_2} \rceil)$.*

In this lemma we use a distributional lifting of obs for intensional distributions, written $\widehat{\text{obs}}$; its definition is identical to $\widetilde{\text{obs}}$ shown in Figure 7 but with the intensional distribution monad $\mathcal{I}$ instead of $\mathcal{D}$.

We are now able to complete the full proof PMTO. The general strategy is to first consider two well-typed source programs which are equal modulo adversary observation. First, these programs are transported to the mixed language, where low-equivalence is established, as well as well-typing. The programs are executed in the mixed semantics, and PMTO for mixed terms is applied, which appeals to type preservation. Due to PMTO for mixed terms, the results will be low-equivalent, and via soundness of low-equivalence, we conclude equality of distributions modulo adversary observation after projection. The final steps are via simulation lemmas, showing that this final

projection lines up with executions of the initial programs in the standard semantics. A detail we have elided until now is that each semantics (standard and mixed) are defined as partial functions. They become total under assumptions of well-typing, which we state and prove as a progress lemma in the appendix.

Theorem 4.7 (PMTO).

If: e_1 *and* e_2 *are source expressions,* $\mathrm{obs}(e_1) = \mathrm{obs}(e_2)$, $\vdash e_1$ *and* $\vdash e_2$
Then: (1) $\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_1)$ *and* $\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_2)$ *are defined*
And: (2) $\widetilde{\mathrm{obs}}(\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_1)) = \widetilde{\mathrm{obs}}(\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_2))$.

Proof.

(1) is by Progress. (2) is by the following:

$$
\begin{array}{lll}
& \mathrm{obs}(e_1) = \mathrm{obs}(e_2) & \\
\Longrightarrow & e_1 \sim e_2 & \wr \text{ by simple induction } \int \\
\Longrightarrow & \overline{\mathrm{nstep}}(N, \varnothing, e_1) \approx_{\sim} \overline{\mathrm{nstep}}(N, \varnothing, e_2) & \wr \text{ by PMTO (Mixed) } \int \\
\Longrightarrow & \widetilde{\mathrm{obs}}(\hat{\lceil}\overline{\mathrm{nstep}}(N, \varnothing, e_1)\hat{\rceil}) \approx_{=} \widetilde{\mathrm{obs}}(\hat{\lceil}\overline{\mathrm{nstep}}(N, \varnothing, e_2)\hat{\rceil}) & \wr \text{ by Low-equivalence Soundness } \int \\
\Longrightarrow & \widetilde{\mathrm{obs}}(\mathrm{nstep}_{\mathcal{I}}(N, \varnothing, e_1)) \approx_{=} \widetilde{\mathrm{obs}}(\mathrm{nstep}_{\mathcal{I}}(N, \varnothing, e_2)) & \wr \text{ by Intensional Simulation } \int \\
\Longrightarrow & \widetilde{\mathrm{obs}}(\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_1)) = \widetilde{\mathrm{obs}}(\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_2)) & \wr \text{ by Ground Truth Simulation } \int
\end{array}
$$

□

The full details of this proof are spelled out in the appendix in the form of 12 figures and 49 lemmas.

5 CASE STUDY: TREE-BASED ORAM

We have implemented an interpreter and type checker for $\lambda_{\mathbf{obliv}}$ in which we have programmed a modern tree-based ORAM [Shi et al. 2011] and a probabilistic oblivious stack [Wang et al. 2014]. The remainder of this section describes our ORAM code; oblivious stacks and various other details are given in the Appendix.

A tree-based ORAM [Shi et al. 2011; Stefanov et al. 2013; Wang et al. 2015] has two parts: a tree-like structure for storing the actual data blocks, and a *position map* that maps logical data block indexes to *position tags* that indicate the block's position in the tree. In the simplest instantiation, an ORAM of size N has a position map of size N which is hidden from the adversary; e.g., it could be stored in on-chip memory in a processor-based deployment of ORAM (see Section 2.1). The position tags mask the relationship between a logical index and the location of its corresponding block in the tree. As blocks are read and written, they are shuffled around in the data structure, and their new locations are recorded in the position map. The position map need not be hidden on chip; rather, much of it can be stored recursively in ORAM itself, reducing the space overhead on the client. As such, the tree part that contains the data blocks is called a *non-recursive ORAM* (or NORAM), and the full ORAM with the position map stored within it is called a *recursive ORAM*. We discuss each part in turn.

To support implementing Tree ORAM, our implementation extends $\lambda_{\mathbf{obliv}}$ in various ways. Three noteworthy extensions are (natural) numbers, arrays, and records. Natural number literals are written n_ℓ. We also support an expression $\texttt{rnd}^{\rho}()$ which is like $\texttt{flip}^{\rho}()$ except it produces a uniform random natural number. Natural numbers and random naturals can be encoded as fixed-width tuples of `bitv` and `flipv` respectively.

Arrays behave and are typed similarly to the references of $\lambda_{\mathbf{obliv}}$. Like references, arrays have kind `U` (universal), meaning they are not treated affinely. Arrays support three operations. First, a $\texttt{length}(a)$ operation which returns the length of the array as a $\texttt{natv}_P$. Second and third, $\texttt{read}[n](a)$

and $\texttt{write}[n](a, e)$ which are like the corresponding operations on references, except that they take a natural number index. We will typically write these operations as $a[n]$ and $a[n] \leftarrow e$ respectively. The typing rules require that the index into the array be public, $\texttt{nat}_P$. When the contents of the array are $\texttt{A}$ (affine), only the simultaneous read/write operation is well-typed, since reading out the contents of the array slot will consume them, and so they must be replaced with something fresh. An array of length N can be encoded as an an N-tuple of references, using nested if-statements to access the correct index.

A record's kind is the join of the kinds of its fields. In other words, if it contains any affine fields then it is treated affinely. Records support field accessor notation, $r.x$, which only consumes the field x rather than consuming all of r (assuming x is affine). As such, other field accesses such as $r.y$ are allowed but any subsequent access $r.x$ would be a type error.

Our implementation supports *region polymorphism* wherein we may specify region variables and then substitute concrete regions for them so long as the substituted regions satisfy programmer-specified ordering constraints. We do not support data polymorphism, although we believe it is a delicate but standard extension.

In what follows, we write $\texttt{nat}_S$ to be the type of a secret number in region $\bot$; $\texttt{nat}_P$ for the type of a public number in $\bot$; $\texttt{nat}^R$ for the type of a secret number in the polymorphic region R. We also write $\texttt{rnd}^R$ to be the type of a random natural number in the polymorphic region R.

5.1 Non-recursive ORAM (NORAM)

Data definition. The definition of a tree-based NORAM is as follows:

```
type (R, R') noram = ((R, R') bucket) array
type (R, R') bucket = ((R, R') block) array
type (R, R') block  = { is_dummy : R bit ; idx : R bit ; tag : R bit ; data : (R' rnd) * (R' rnd) }
                      where R ⊏ R'
```

A noram is an array of $2N - 1$ buckets which represents a complete tree in the style of a heap data structure: for the node at index $i \in \{0, \ldots, 2N - 2\}$, its parents, left child, and right child correspond to the nodes at index $(i - 1)/2$, $2i + 1$, and $2i + 2$, respectively. Each bucket is an array of blocks, each of which is a record where the data field contains the data stored in that bucket. The other three components of the block are secret; they are (1) the is_dummy bit indicating if the block is dummy (empty) or not; (2) the index (idx) of the block; and (3) the position tag of the block. Note that the bucket type, ignoring the position tag, is essentially a *Trivial ORAM*, as discussed below.

We choose type (R' rnd) * (R' rnd) for the data portion to illustrate that affine values can be stored in the NORAM, and to set up our implementation of full, recursive ORAM, next.

Operations. norams do not implement read and write operations directly; these are implemented using two more primitive operations called noram_readAndRemove (or noram_rr, for short) and noram_add. The former reads the designated element from noram and also removes it, while the latter adds the designated element (overwriting any version already there). The code for noram_rr is given below; we explain it just afterward.

```
1  let rec trivial_rr_h (troram : (R, R') bucket) (idx : R natS) (i : natP) (acc : (R, R') block) : (R, R')
       block =
2    if i = length(troram) then acc
3    else
4      (* read out the current block *)
5      let curr = bucket[i] ← (dummy_block ()) in
6      (* check if the current block is non-dummy, and it's index matches the queried one *)
7      let swap : R bit = !curr.is_dummy && curr.idx = idx in
8      let (curr, acc) = mux(swap, acc, curr) in
9      (* when swap is false, this equivalent to writing the data back; otherwise, acc
10        stores the found block and is passed into the next iteration *)
11     let _ = bucket[i] ← curr in
```

```
12      trivial_rr_h troram idx (i + 1) acc
13
14  let trivial_rr (troram : (R, R') bucket) (idx : R natS) : (R' rnd) * (R' rnd) =
15    let ret : block = trivial_rr_h troram idx 0 (dummy_block ()) in
16    ret.data
17
18  let rec noram_rr_h (noram : (R, R') noram) (idx : R natS) (tag : natP) (level : natP) (acc : (R, R') block)
          : (R, R') block =
19    (* compute the first index into the bucket array at depth level *)
20    let base : natP = (pow 2 level) - 1 in
21    if base >= length(noram) then acc
22    else
23      let bucket_loc : natP = base + (tag & base) in (* the bucket on the path to access *)
24      let bucket = noram[bucket_loc] in
25      let acc = trivial_rr_h bucket idx 0 acc in
26      noram_rr_h noram idx tag (level + 1) acc
27
28  let noram_rr (noram : (R, R') noram) (idx : R natS) (tag : natP) : (R' rnd) * (R' rnd) =
29    let ret = noram_rr_h noram idx tag 0 (dummy_block ()) in
30    ret.data
```

The noram_rr function takes the noram as its first argument, and the index idx of the desired element as its second. The tag argument is the position tag, which identifies a path through the noram binary tree along which the indexed value will be stored, if present. This argument has type natP because it (or derivatives of it) will be used index the arrays that make up the NORAM, and these indexes are adversary-visible. The noram_rr function works by calling noram_rr_h which recursively works its way down the identified path. It maintains an accumulator, acc : (R, R') block, over the course of the traversal. Initially, acc is a dummy block. The dummy_block () is a function call rather than a constant because the block record contains data: (R' rnd) * (R' rnd). This member of the record must be generated fresh for each new block, since its contents are treated affinely. Each recursive call to noram_rr_h moves to a node the next level down in the tree, as determined by the tag. At each node, it reads out the bucket array, which as mentioned earlier is essentially a Trivial ORAM. The trivial_rr function calls trivial_rr_h to iterate through the entire bucket, to obliviously read out the desired block, if present.

Notice that we are using arrays with both affine and non-affine (universal) contents in this code. The noram type has contents which are kind U, since the type of its contents are arrays. As such, we can read from noram without writing a new value (line 24). However, the (R, R') bucket type has contents which are kind A, since the type of its contents are tuples which contain type R' rnd. So, when we index into members of values of type (R, R') bucket we must write a dummy block (line 5).

This algorithm for noram_rr will access $\log N$ buckets (where N is the number of buckets in the noram), and each bucket access causes a trivial_rr which takes time b where b is the size of each bucket. Therefore, the noram_rr operation above takes time $O(b \log N)$. In the state-of-the-art ORAM constructions, such as Circuit ORAM [Wang et al. 2015], b can be parameterized as a constant (e.g., 4), which renders the overall time complexity of noram_rr to be $O(\log N)$. This is asymptotically faster than implementing the entire ORAM as a Trivial ORAM, which takes time $O(N)$.

The noram_add routine has the following signature:

```
val noram_add : (R, R') noram → (idx : R natS) → (tag : R natS) → (data : (R' rnd) *
      (R' rnd)) → unit
```

Like the noram_rr operation, it takes an index and a position tag, but here the position tag is secret, since it will not be examined by the algorithm. In particular, noram_add simply stores a block consisting of the dummy bit, index, position tag, and data into the root bucket of the noram. It does this as a Trivial ORAM operation: It iterates down the array similarly to trivial_rr above, but stores the new block in the first available slot.

To avoid overflowing the root's bucket due to repeated noram_adds, a tree-based ORAM employs an additional eviction routine, usually called after both noram_add and noram_rr, to move blocks closer

to the leaf buckets. This routine should also maintain the key invariant: each data block should always reside on the path from the root to the leaf corresponding to its position tag. Different tree-based ORAM implementations differ only in their choices of b and the eviction strategies. One simple eviction strategy picks two random nodes at each level of the tree, reads a single non-empty block from each chosen node's bucket, and then writes that block one level further down either to the left or right according to the position tag; a dummy block is written in the opposite direction to make the operation oblivious.

5.2 Full Recursive ORAM

To use noram to build a full ORAM, we need a way to get the position tag for reads and writes. Such tags are stored in a *position map*. Such a map PM maps any index $i \in 0..N$, where N is the capacity of ORAM, to randomly generated position tag. A full ORAM read at index i works by looking up the tag in the position map, performing an noram_rr to read and remove the value at i, and noram_adding back that value (updating the position map with a fresh tag). A write to i requires a noram_rr to remove the old value at i, and an noram_add to write the new value (again updating the position map). In doing so, accessing the same index twice will assign two uniformly independently sampled random position tags to the same data block, so that the adversary cannot learn any information about whether the two accesses are corresponding to the same or different indexes.

As mentioned above, one way to implement the position map is to just use a regular array stored in hidden memory, e.g., on-chip in a secure processor deployment of ORAM. However, this is not possible for MPC-based deployments, the adversary can observe the access pattern on the map itself. To avoid this problem, we can use another smaller Tree-based ORAM to store PM. In particular, the ORAM PM has N/c blocks, where each block contains c elements, where $c \geq 2$ is a parameter of the ORAM. Therefore, PM also contains another position map PM $'$, which can be stored as an ORAM of capacity N/c^2. Such a construction can continue recursively, until the position map at the bottom is small enough to be constructed as a Trivial ORAM. Therefore, there are $\log_c N$ norams constructed to store those position maps, and thus the overall runtime of a recursive ORAM is $\log_c N$ times of the runtime of a noram. In the following, we implement the ORAM with $c = 2$.[3]

A full ORAM thus has the type oram, given below.

```
type oram R R' = ((noram R R') array) * (bucket R R')
```

In short, an ORAM is a sequence of norams, where the first in the sequence contains the actual data, and the remainder serve as the position map, which eventually terminate with a trivial ORAM (the bucket at the end). The main idea is to think of the position map as essentially an array of size N but "stored" as a 2-D array: `array[N/2][2]`. In doing so, to access PM[idx], we essentially access PM[$idx/2$][idx mod 2].

We implement the tree_rr as a call to the function tree_rr_h, which takes an additional public level argument, to indicate at which point in the list of orams to start its work; it's initially called with level 0.

```
let rec tree_rr_h (oram: oram) (idx: nat S) (level: nat P): rnd R' * rnd R' =
  let (norams, troram) = oram in
  let levels: nat P = length(norams) in
  if level >= levels then trivial_rr troram idx
  else
    let (r0, r1): rnd R' * rnd R' = tree_rr_h oram (idx / 2) (level + 1) in
    let (r0', tag) = mux(idx % 2 = 0, rnd, r0) in
    let (r1', tag) = mux(idx % 2 = 1, tag, r1) in
    let _ = tree_add_h oram (idx / 2) (level + 1) (r0', r1') in
    noram_rr norams[level] idx (castP tag)

```

[3]We present a readable, almost-correct version of the code first, and then clarify the remaining issue at the end.

```
12  let tree_rr (oram: oram) (idx: nat S): rnd R' * rnd R' =
13    tree_rr_h oram idx 0
```

Line 4 checks whether we have hit the base case of the recursion, in which case we lookup idx in the troram, returning back a rnd r1 * rnd r1 pair. Otherwise, we enter the recursive case. Here, line 6 essentially reads out PM[$idx/2$], and lines 7 and 8 obliviously read out PM[$idx/2$][idx mod 2] into tag, replacing it with a freshly generated tag, to satisfy the affinity requirement. Finally, line 8 writes the updated block PM[$idx/2$] back (i.e., (r0', r1')), using an analogous tree_add_h routine, for which a level can be specified. Finally, line 9 reveals the retrieved position tag for index idx, so that it can be passed into the noram_rr routine of noram. Level 0 corresponds to the actual data of the ORAM, which is returned to the client.

The tree_add routine is similar so we do not show it all. As with tree_rr it recursively adds the corresponding bits of the position tag into the array of norams. At each level of the recursion there is a snippet like the following:

```
1  let new_tag: rnd R' = rnd in
2  let sec_tag = castS new_tag in (* does NOT consume new_tag *)
3  let (r0, r1): rnd R' * rnd R' = tree_rr_rec oram (idx / 2) (level + 1) in
4  let r0', tag = mux (idx % 2 = 0, new_tag, r0) in  (* replaces with new tag *)
5  let r1', tag = mux (idx % 2 = 1, tag, r1) in
6  let _ = tree_add_rec oram (idx / 2) (level + 1) (r0', r1') in
7  noram_add norams[level] idx sec_tag data (* adds to Tree ORAM *)
```

Lines 1 and 2 generate a new tag, and make a secret copy of it. The new tag is then stored in the recursive ORAM—lines 3–5 are similar to tree_add_rec but replace the found tag with new_tag, not some garbage value, at the appropriate level of the position map (line 6). Finally, sec_tag is used to store the data in the appropriate level of the noram.

The astute reader may have noticed that the code snippet for add will not type check. In particular, the sec_tag argument has type nat R' but noram_add requires it to have type nat R. This is because the position tags for the noram at level are stored as the data of the noram at level + 1, and these are in different regions. We cannot put them in the same region because we require a single noram's metadata and data to reside in different regions. We can solve this problem by having each level use the opposite pair of regions as the one above it. This solves the type error and does not compromise the noram independence requirement.

Basically, tree_rr_h will have to operate two levels at a time in order to satisfy the type checker, but the basic logic will be the same. In addition, tree_rr_h and tree_rr require the type of idx to be nat S (rather than nat R for some region variable R). Recall that nat S is really nat with the $\emptyset$ region. Because $\emptyset \subseteq$ R and $\emptyset \subseteq$ R', idx can be passed to rr at both of the unrolled levels, one of which requires it to be R and the other, R'.

Oblivious Stacks. Other oblivious data structures can be built in $\lambda_{\textbf{obliv}}$, and on top of noram in particular. Appendix A.1 presents a development of probabilistic oblivious stacks, along with some additional technical challenges they present.

6 RELATED WORK

Lampson first pointed out various covert, or "side," channels of information leakage during a program's execution [Lampson 1973]. Defending against side-channel leakage is challenging. Previous works have attempted to thwart such leakage from various angles:

- Processor architectures that mitigate leakage through timing [Kocher et al. 2004; Liu et al. 2012], power consumption [Kocher et al. 2004], or memory-traces [Fletcher et al. 2014; Liu et al. 2015a; Maas et al. 2013; Ren et al. 2013].
- Program analysis techniques that formally ensure that a program has bounded or no leakage through instruction traces [Molnar et al. 2006], timing channels [Agat 2000; Molnar et al.

2006; Russo et al. 2006; Zhang et al. 2012, 2015], or memory traces [Liu et al. 2015a, 2013, 2014].

- Algorithmic techniques that transform programs and algorithms to their side-channel-mitigating or side-channel-free counterparts while introducing only mild costs – e.g., works on mitigating timing channel leakage [Askarov et al. 2010; Barthe et al. 2010; Zhang et al. 2011], and on preventing memory-trace leakage [Blanton et al. 2013; Eppstein et al. 2010; Goldreich 1987; Goldreich and Ostrovsky 1996; Goodrich et al. 2012; Shi et al. 2011; Stefanov et al. 2013; Wang et al. 2015, 2014; Zahur and Evans 2013].

Often, the most effective and efficient is through a comprehensive co-design approach combining these areas of advances – in fact, several aforementioned works indeed combine (a subset of) algorithms, architecture, and programming language techniques [Fletcher et al. 2014; Liu et al. 2015a; Ren et al. 2013; Zhang et al. 2012, 2015].

Our core language not only generalizes a line of prior works on timing channel security [Agat 2000], program counter security [Molnar et al. 2006], and memory-trace obliviousness [Liu et al. 2015a, 2013, 2014], but also provides the first distinct core language that captures the essence of memory-trace obliviousness without treating key mechanisms—notably, the use of randomness—as a black box. Thus we can express state-of-the-art algorithmic results and formally reason about the security of their implementations, thus building a bridge between algorithmic and programming language techniques.

ObliVM [Liu et al. 2015b] is a language for programming oblivious algorithms intended to be run as secure multiparty computations [Yao 1986]. Its type system also employs affine types to ensure random numbers are used at most once. However, there it provides no mechanism to disallow constructing a non-uniformly distributed random number. When such random numbers are generated, they can be distinguished by an attacker from uniformly distributed random numbers when being revealed. Therefore, the type system in ObliVM does not guarantee obliviousness. $\lambda_{\mathbf{obliv}}$'s use of probability regions enforces that all random numbers are uniformly random, and thus eliminates this channel of information leakage. Moreover, we prove that this mechanism (and the others in $\lambda_{\mathbf{obliv}}$) are sufficient to prove PMTO. For programs which are PMTO but temporarily violate uniformity, probabilistic program verification [Barthe et al. 2018] provides a mechanism for discharging proof obligations (of uniformity) introduced by unsafe casts.

Our work belongs to a large category of work that aims to statically enforce *noninterference*, e.g., by typing [Sabelfeld and Myers 2006; Volpano et al. 1996]. Liu et al. [2015a, 2013] developed a type system that ensures programs are MTO. In their system, types are extended to indicate where values are allocated; as per our above example data can be public or secret, but can also reside in ORAM. Trace events are extended to model ORAM accesses as opaque to the adversary (similar to the Dolev-Yao modeling of encrypted messages [Dolev and Yao 1981]): the adversary knows that an access occurred, but not the address or whether it was a read or a write. Liu et al's type system enforces obliviousness of deterministic programs that use (assumed-to-be-correct) ORAM.

Our probabilistic memory trace obliviousness property bears some resemblance to probabilistic notions of noninterference. Much prior work [Ngo et al. 2014; Russo and Sabelfeld 2006; Sabelfeld and Sands 2000; Smith 2003] is concerned with how random choices made by a thread scheduler could cause the distribution of visible events to differ due to the values of secrets. Here, the source of nondeterminism is the (external) scheduler, rather than the program itself, as in our case. Smith and Alpízar [2006, 2007] consider how the influence of random numbers may affect the likelihood of certain outcomes, mostly being concerned with termination channels. Their programming model is not as rich as ours, as a secret random number is never permitted to be made public; such an ability is the main source of complexity in $\lambda_{\mathbf{obliv}}$, and is crucial for supporting oblivious algorithms.

Some prior work aims to quantify the information released by a (possibly randomized) program (e.g., Köpf and Rybalchenko [2013]; Mu and Clark [2009]) according to entropy-based measures. Work on verifying the correctness of differentially private algorithms [Barthe et al. 2013; Zhang and Kifer 2017] essentially aims to bound possible leakage; by contrast, we enforce that *no* information leaks due to a program's execution.

7 CONCLUSIONS

This paper has presented $\lambda_{\textbf{obliv}}$, a core language suitable for expressing computations whose execution should be oblivious to a powerful adversary who can observe an execution's trace of instructions and memory accesses, but not see private values. Unlike prior formalisms, $\lambda_{\textbf{obliv}}$ can be used to express probabilistic algorithms whose security depends crucially on the use of randomness. To do so, $\lambda_{\textbf{obliv}}$ tracks the use of randomly generated numbers via a substructural (affine) type system, and employs a novel concept called *probability regions*. The latter are used to track a random number's probabilistic (in)dependence on other random numbers. We have proved that together these mechanisms ensure that a random number's revelation in the visible trace does not perturb the distribution of possible events so as to make secrets more likely. We have demonstrated that $\lambda_{\textbf{obliv}}$'s type system is powerful enough to accept sophisticated algorithms, including forms of oblivious RAMs. Such data structures were out of the reach of prior type systems. As such, our proof-of-concept implementations represent the first automated proofs that these algorithms are secure. Ultimately we hope to integrate our ideas into ObliVM [Liu et al. 2015b], a state-of-the-art compiler for oblivious algorithms, and thereby ensure that well-typed programs are indeed secure.

ACKNOWLEDGMENTS

We thank Aseem Rastogi and Kesha Heitala for comments on earlier drafts of this paper, and Elaine Shi for helpful suggestions and comments on the work while it was underway. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1563722, CNS-1314857, and CNS-1111599, and by DARPA under contracts FA8750-15-2-0104 and FA8750-16-C-0022. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Johan Agat. 2000. Transforming out Timing Leaks. In *POPL*.

Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. 2010. Predictive black-box mitigation of timing channels. In *CCS*.

Henry G. Baker. 1992. Lively Linear Lisp: &Ldquo;Look Ma, No Garbage!&Rdquo;. *SIGPLAN Not.* 27, 8 (Aug. 1992), 89–98. https://doi.org/10.1145/142137.142162

Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. An Assertion-Based Program Logic for Probabilistic Programs. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 117–144.

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2013. Probabilistic Relational Reasoning for Differential Privacy. *ACM Trans. Program. Lang. Syst.* 35, 3 (2013), 9:1–9:49.

Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. 2010. Security of multithreaded programs by compilation. *ACM Transactions on Information and System Security (TISSEC)* 13, 3 (2010), 21.

Marina Blanton, Aaron Steele, and Mehrdad Alisagari. 2013. Data-oblivious Graph Algorithms for Secure Computation and Outsourcing. In *ASIA CCS*.

David Brumley and Dan Boneh. 2003. Remote Timing Attacks Are Practical. In *USENIX Security*.

D. Dolev and A. C. Yao. 1981. On the Security of Public Key Protocols. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science (SFCS)*.

Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *PLDI*.

David Eppstein, Michael T. Goodrich, and Roberto Tamassia. 2010. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *GIS*.

Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical computer science* 103, 2 (1992), 235–271.

Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk, Omer Khan, and Srinivas Devadas. 2014. Suppressing the Oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*.

Michèle Giry. 1982. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, B. Banaschewski (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 68–85.

J.A. Goguen and J. Meseguer. 1982. Security policy and security models. In *IEEE S & P*.

O. Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*.

O. Goldreich, S. Micali, and A. Wigderson. 1987. How to play ANY mental game. In *STOC*.

Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *J. ACM* (1996).

Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. 2012. Data-Oblivious Graph Drawing Model and Algorithms. *CoRR* abs/1209.0756 (2012).

Matt Hoekstra. 2015. Intel SGX for Dummies (Intel SGX Design Objectives). https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx.

Mohammad Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Network and Distributed System Security Symposium (NDSS)*.

Paul Kocher, Ruby Lee, Gary McGraw, and Anand Raghunathan. 2004. Security As a New Dimension in Embedded System Design. In *Proceedings of the 41st Annual Design Automation Conference (DAC '04)*. 753–760. Moderator-Ravi, Srivaths.

Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*.

Boris Köpf and Andrey Rybalchenko. 2013. Automation of quantitative information-flow analysis. In *Formal Methods for Dynamical Systems*.

Butler W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* (1973).

Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015a. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *ASPLOS*.

Chang Liu, Michael Hicks, and Elaine Shi. 2013. Memory Trace Oblivious Program Execution. In *CSF*.

Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. 2014. Automating Efficient RAM-Model Secure Computation. In *IEEE S & P*.

Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015b. ObliVM: A Programming Framework for Secure Computation. In *IEEE S & P*.

Isaac Liu, Jan Reineke, David Broman, Michael Zimmer, and Edward A. Lee. 2012. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *ICCD*.

Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. 2013. Phantom: Practical Oblivious Computation in a Secure Processor. In *CCS*.

David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2006. The Program Counter Security Model: Automatic Detection and Removal of Control-flow Side Channel Attacks. In *ICISC*.

Chunyan Mu and David Clark. 2009. An abstraction quantifying information flow over probabilistic semantics. In *Workshop on Quantitative Aspects of Programming Languages (QAPL)*.

Tri Minh Ngo, Mariëlle Stoelinga, and Marieke Huisman. 2014. Effective verification of confidentiality for multi-threaded programs. *Journal of computer security* 22, 2 (2014).

Norman Ramsey and Avi Pfeffer. 2002. Stochastic Lambda Calculus and Monads of Probability Distributions. In *POPL*.

Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. 2013. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*.

Alejandro Russo, John Hughes, David A. Naumann, and Andrei Sabelfeld. 2006. Closing Internal Timing Channels by Transformation. In *Annual Asian Computing Science Conference (ASIAN)*.

Alejandro Russo and Andrei Sabelfeld. 2006. Securing interaction between threads and the scheduler. In *CSF-W*.

A. Sabelfeld and A. C. Myers. 2006. Language-based Information-flow Security. *IEEE J.Sel. A. Commun.* 21, 1 (Sept. 2006).

Andrei Sabelfeld and David Sands. 2000. Probabilistic noninterference for multi-threaded programs. In *CSF-W*.

Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *ASIACRYPT*.

Geoffrey Smith. 2003. Probabilistic noninterference through weak probabilistic bisimulation. In *CSF-W*.

Geoffrey Smith and Rafael Alpízar. 2006. Secure Information Flow with Random Assignment and Encryption. In *Workshop on Formal Methods in Security (FMSE)*.

Geoffrey Smith and Rafael Alpízar. 2007. Fast Probabilistic Simulation, Nontermination, and Secure Information Flow. In *PLAS*.

Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM – an Extremely Simple Oblivious RAM Protocol. In *CCS*.

G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ICS*.

David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural support for copy and tamper resistant software. *SIGOPS Oper. Syst. Rev.* 34, 5 (Nov. 2000).

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2-3 (Jan. 1996).

Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *CCS*.

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

Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *FOCS*.

Samee Zahur and David Evans. 2013. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In *S & P*.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2011. Predictive Mitigation of Timing Channels in Interactive Systems. In *CCS*.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2012. Language-based Control and Mitigation of Timing Channels. In *PLDI*.

Danfeng Zhang and Daniel Kifer. 2017. LightDP: Towards Automating Differential Privacy Proofs. In *POPL*.

Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *ASPLOS*.

Xiaotong Zhuang, Tao Zhang, and Santosh Pande. 2004. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. *SIGARCH Comput. Archit. News* 32, 5 (Oct. 2004).

```
1  type Stack = (nat S * rnd R) noram * nat S ref
2  let stackop ((oram,rid_r): Stack) (ispush: bit S) (d: nat S): nat S =
3  let rid = !rid_r in
4  let old_d = read oram rid in
5  let (d', _) = mux ispush d old_d in
6  let (id, _) = mux ispush (rid+1) rid in
7  write oram id d';
8  let (rid', _) = mux ispush (rid+1) (rid−1) in
9  rid_r := rid'; d'
```

Fig. 12. A deterministic oblivious stack built using a full (recursive) ORAM.

```
1  type OStack = (nat S * rnd R) noram * nat S ref * rnd R ref
2  let stackop ((noram,rid_r,pos_r): OStack) (ispush: bit S) (d: nat S) : nat S =
3  let (rid,pos) = !rid_r ,!pos_r in
4  let (rid', pos', d') =
5    if ispush then
6      let (d', _) = noram_rr noram (−1) (castP (rnd())) in
7      let b = (d,pos) in
8      let pos' = rnd() in
9      let _ = noram_add noram (rid+1) (castS pos') b in
10     (rid+1,pos', d')
11   else
12     let (d', pos') = noram_rr noram rid (castP pos) in
13     let b = (d,rnd()) in
14     let _ = noram_add noram (−1) 0 b in
15     (rid−1,pos', d') in
16 rid_r := rid';
17 rpos_r := pos';
18 d'
```

Fig. 13. A probabilistic oblivious stack built using a non-recursive ORAM. (Does not use **mux**, for simplicity.)

A APPENDIX

A.1 Case Study: Oblivious Stacks

This section considers an oblivious data structure, an *oblivious stack*. The goal of an oblivious stack is to hide both its data and which operations (pushes or pops) are taking place—only the total number of operations should be revealed. To do this, we could implement the stack using an ORAM rather than a normal (encrypted) array, and we could merge the code for push and pop so as to mask which operation is taking place (despite knowledge of the PC). Code to do this is shown in the `stackop` function in Figure 12.

In the code, a stack consists of an ORAM of secret numbers and a reference storing the index of the root. Function `stackop` takes a stack, a flag indicating whether the operation is a push or pop, and the value to push, and returns a value. The code reads the value at the root index (line 3). The next line copies that value to `d'` if the operation is pop, or else puts `d` there if it is a push. Line 5 determines the index of the write it will perform on line 6: this index (`id`) is one more than the root index if it's a push and it's the current root index if not. As such, the write on line 7 puts the given value in the next slot in case of a push, or writes back the value at the current root, if it's a pop. Finally, line 8 adjusts the root index, and line 9 returns the result, which is either the popped value or pushed value (if it was a push).

While this code works perfectly well a *probabilistic* version of the stack, using a *non-recursive ORAM* would be more space-efficient. In particular, it will require only $O(L)$ extra space where L is the current size of the stack, whereas this version requires $O(N)$ extra space, where N is the size of the ORAM. To see how, consider that we always access a stack via its head, using the root index. Thus, in the code in Figure 12, the full `oram` internally only ever uses one slot in its position map. Thus we can do better by using an NORAM directly, having the stack manage the position tag of the root. In short, we implement an oblivious stack as a triple comprising a NORAM, the index of

```
1  type OStack = (nat S * rnd R) noram * nat S ref * rnd R ref
2  let stackop ((noram,rid_r,pos_r): OStack) (ispush: bit S) (d: nat S) : nat S =
3  let (rid,pos) = !rid_r,!pos_r in
4  let (rid',pos',d') =
5    let (id, new_rid) = mux(ispush, -1, rid+1) in
6    let (to_cast_p, tmp) = mux(ispush, rnd(), pos) in
7    let (d', pos') = noram_rr noram id (castP to_cast_p) in
8    let (pos', _) = mux(ispush, rnd(), pos') in
9    let b = (d, tmp) in
10   let (pos_S, _) = mux(ispush, pos', 0) in
11   let _ = noram_add noram new_rid (castS pos_S) b in
12   let (ret_rid, _) = mux(ispush, rid_r - 1, rid_r + 1) in
13   (ret_id, pos', d') in
14 rid_r := rid';
15 rpos_r := pos';
16 d'
```

Fig. 14. A probabilistic oblivious stack built using a non-recursive ORAM using **mux**.)

the root element, and its position tag. The latter two act as a kind of pointer into the NORAM. Each block stored in the NORAM contains the data and the position tag of the next block in the stack.

Code implementing the stack following this design is given in Figure 13. Note the code branches on the ispush variable to make it easier to read; the actual implementation must use muxs to conditionally execute each statement in both branches to ensure obliviousness.[4] Line 3 extracts the current root index and position tag. Lines 6–10 handle a push operation. Line 6 first does a "dummy read" from the NORAM; just as we saw with the trivial ORAM add earlier, using index −1 results in a dummy block being returned (the position tag argument is unimportant in this case). Line 7 constructs a new block b to push: it consists of the given data d paired with the current root's position tag pos, thus creating a "pointer" to that block. We then generate a fresh position tag pos' for this (the new root's) block, add the block to the noram. Random numbers generated by rnd() have type rnd R; the coercion **castP** ascribes a random number the type int P (per line 6), while **castS** gives it type int S (line 9); we explain these constructs in the next subsection. The new root index (the old one plus one), the root's tag, and the dummy block passed in are returned on line 10. Lines 12–15 handle a pop. Here, the first rr does real work, extracting the block that corresponds to the root index and position tag. We then generate a dummy block to "add" to the ORAM. The updated root index (the old one minus one), its position tag (returned by the rr) and the fetched block are returned. The full mux version is provided in Figure 14.

This version of an oblivious stack performs better than the version from Figure 12. The space overhead, due to the added pointers at the root and within the ORAM, is $O(L)$ where L is the size of the current stack, not the size N of the ORAM. The running time is still $O(\log N)$. Obliviousness is a direct corollary of implementing our stack on $\lambda_{\textbf{obliv}}$: Because we have labeled the stack's contents and root as secret, as well as the choice of operation, nothing can be learned about any of them when observing the event trace.

A.2 The Limits of Syntactic Uniformity Enforcement

Unfortunately, we cannot directly typecheck an implementation of oblivious stacks. To see why, consider the type of block of Section 5.1. In this type we assume that the position tag field is in region R and that this region is independent of R' the region associated with random values stored inside the NORAM. However, in the code for oblivious stacks, we are storing random numbers in region R' that we will later use as the position tag for subsequent operations. So, it is clearly not the case that the independence constraint is satisfied. So, what if we make the following change to the type for blocks:

[4]Notice that the structure of both branches is roughly parallel, which makes converting to the use of **mux**es straightforward.

```
type block  R R' = { is_dummy : bit R ; idx : nat R ; tag : nat R' ; data : (rnd R' * rnd R') }
                   where R ⊥ R'
```

where we place the position tag in region R' instead. This almost works – since the tag argument is public in noram_rr and noram_add operations only mux on the index (in the trivial write to the root bucket). However, the eviction procedure will not typecheck with the position tag at region R'. This is because the eviction procedure performs a **mux** on the secret position tag (in R') to decide where to evict a block (with the data component also in R'). Thus, the independence requirement for the type rule is not met. The fundamental issue is that we are storing position tags in the ORAM, so the region associated with position tags *of* the ORAM are the same as the regions of the data *in* the ORAM.

Our type system rejects the mux in the eviction procedure (when position tags are typed at region R') because it *does* yield random values which are not uniformly distributed. However, this violation is actually a false-positive. By the time these random values are revealed to the adversary, their uniformity is re-established. This is obviously the case, because if the ORAM truly implements a map, the result of reading front the ORAM on line 12 of stackop will yield the same value that was placed into it. This value was a fresh random value, which is uniformly distributed. The following simple, pathological case illustrates the issue:

```
let s = true S in
let (r0, r1) = (flip R0, flip R1) in
let r0_s = castS r0 in
let g = s && r0_s in
let (v1, v2) = mux (g, r0, r1) in
let (v, _) = mux (g, v1, v2) in
castP v
```

This example flips two coins and uses them as the arguments to the **mux** on line 5. Since the guard of this **mux** depends on the value of r0 the resulting values v1, v2 are not uniformly distributed. Indeed, the type system would reject the **mux** on line 5. However, the **mux** on line 6 yields another value v which is again uniformly distributed and thus safe to reveal. The takeaway here is that a sequence of appropriate **mux**es can temporarily perturb and then re-establish the uniformity of a random value before revealing it to the adversary. Since our type system forces random values to be uniform *everywhere*, we cannot typecheck instances like this.

Fortunately, we can easily handle this case with the use of unsafe casts – castNU and castU for cast "non-uniform" and cast "uniform" respectively. This allows a value to be labeled as intentionally non-uniform. While a value is marked as non-uniform, the **mux** operations over these values will not be checked for independence. However, it is the programmers responsibility to ensure at the point that it is casted back into a random value (using castU) that it is truly a random value. For example, we could patch the code above as follows:

```
let s = true S in
let (r0, r1) = (flip R0, flip R1) in
let r0_s = castS r0 in
let g = s && r0_s in
let (v1, v2) = mux (g, (castNU r0), (castNU r1)) in
let (v, _) = mux (g, v1, v2) in
castP (castU v)
```

It is important to note that, if casts are used correctly, then PMTO is preserved. In other words, if all instances of castU are used on values which are truly uniformly distributed then the type system ensures that the program is PMTO. These uniformity obligations can be verified manually, or using external tools [Barthe et al. 2018].

Using this simple extension we can insert a single castNU prior to the **mux** in the eviction procedure. We can then introduce a castU at the point when the position tag is read from the ORAM on line 12 of stackop in Figure 14.