A Language for Probabilistically Oblivious Computation

DAVID DARAIS[*], University of Vermont, USA
IAN SWEET, University of Maryland, USA
CHANG LIU[*], Citadel Securities, USA
MICHAEL HICKS, University of Maryland, USA

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 observed distribution of visible events. Probability regions support reasoning about *probabilistic correlation and independence* 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 expressive enough to typecheck advanced tree-based oblivious RAMs.

CCS Concepts: • **Security and privacy** → **Logic and verification**.

Additional Key Words and Phrases: Oblivious Computation; Type Systems; Probability; Noninterference.

ACM Reference Format:
David Darais, Ian Sweet, Chang Liu, and Michael Hicks. 2020. A Language for Probabilistically Oblivious Computation. *Proc. ACM Program. Lang.* 4, POPL, Article 50 (January 2020), 31 pages. https://doi.org/10.1145/3371118

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] (as made famous by recent Spectre and Meltdown attacks [Kocher et al. 2019; Lipp et al. 2018; Van Bulck et al. 2018]), among other "side" channels

[*] Work carried out in part while at the University of Maryland

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

2475-1421/2020/1-ART50
https://doi.org/10.1145/3371118

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 store code and data in *oblivious RAM* (ORAM) [Maas et al. 2013; Suh et al. 2003]. First proposed by Goldreich [1987] and 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 asymptotic performance improvements. Wang et al. [2014] explored how to build *oblivious data structures* (ODSs), such as queues or stacks, that are more efficient than their standard counterparts implemented on top of ORAM. In followup work, Liu et al. [2015b]; oblivm-www [2019] devised ObliVM, a programming language for implementing such oblivious data structures, including ORAMs themselves. 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 standard language with primitives for generating and using uniformly distributed random numbers. 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 a design which achieves soundness 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 hidden choice 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. Affinity prevents revealing the same number twice, which is problematic because a second revelation is not uniformly distributed when conditioned on observing the first. 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 and then decide to reveal the original random number based on the value of the copy. The type system prevents such behavior 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 $\lambda_{\mathbf{obliv}}$ 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. Full proofs may be found in the supplemental report [Darais et al. 2019].

$\lambda_{\mathbf{obliv}}$ is expressive enough to type check interesting algorithms. Section 5.2 presents the implementation of a tree-based, non-recursive ORAM (NORAM) that type checks in a straightforward extension of $\lambda_{\mathbf{obliv}}$; we have implemented a type checker for this extension. Such an NORAM is a key component of state-of-the-art ORAM implementations [Shi et al. 2011; Stefanov et al. 2013; Wang et al. 2015] and other oblivious data structures [Wang et al. 2014], and to our knowledge ours is the first implementation automatically verified to be oblivious. Section 5.3 shows that *recursive* ORAM, built on NORAM, is also possible but requires a few more advanced (but standard) language features we have not implemented, including region polymorphism, recursive and variant types, and existential quantification. We have also experimented with implementing oblivious data structures using our NORAM; the supplemental report presents *oblivious stacks* (ostacks) in detail. Unfortunately, $\lambda_{\mathbf{obliv}}$'s strict ordering on probability regions is too strong, so the complete ostack implementation will not typecheck. An interesting future direction would be to apply the approach of Zhang et al. [2019a] to integrate $\lambda_{\mathbf{obliv}}$'s type system with a general logic, such as that by Barthe et al. [2020], which can be be used to justify that omitting the probability region order check is (locally) safe. We elaborate in Section 6 when we discuss related work and make the case that $\lambda_{\mathbf{obliv}}$ subsumes previous work on type system design for oblivious computation. Our type checker and all code examples are online at https://github.com/plum-umd/oblivml.

2 OVERVIEW

This section first presents the threat model. Then it discusses *deterministic* oblivious execution, considered by prior work. Finally, it sketches our novel type system for enforcing *probabilistic* oblivious execution, which we develop in full in the rest of the paper.

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, and can observe during execution the PC, the contents of memory, and memory access patterns. Some *secret* memory contents may be encrypted (while *public* memory is not) but all addresses used to access memory are still visible.

Consider an untrusted cloud provider using a secure processor, like SGX [Hoekstra 2015]. Reads/writes from/to 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 setting 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—the input values from each party—are kept hidden from both using secret sharing.

By handling such a strong adversary, our techniques can also handle adversaries with fewer capabilities, 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.

```
B[0] ← s0
B[1] ← s1
 ...
let s = ... // secret bit
let r = B[s] // leaks s
   // via address trace
```

(a) Leaky program

```
B[0] ← s0
B[1] ← s1
 ...
let s = ... // secret bit
let s0' = B[0]
let s1' = B[1]
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

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 addresses of public and secret values that are read and written. (Secret *values* are not visible in the trace.) Under this model, MTO means that running *low-equivalent* input states $\sigma_1; e_1$ and $\sigma_2; e_2$ will produce the exact same memory trace, along with low-equivalent output states. Two states are low equivalent if they agree on the code and public values (but may differ on secret values). More formally, MTO states that if $\sigma_1; e_1 \sim \sigma_2; e_2$ and $\sigma_1; e_1 \longrightarrow^{t} \sigma_1'; e_1'$ then there exists $\sigma_2'; e_2'$ s.t. $\sigma_2; e_2 \longrightarrow^{t} \sigma_2'; e_2'$ and $\sigma_1'; e_1' \sim \sigma_2'; e_2'$, where $\sim$ 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 just 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 address was used (in trace t), they can 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 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. The program in Figure 1(c) illustrates the idea, hinting 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 reads B only once, not 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 recursive ORAMs with access times of $O(log_c\ N)$ for some c (rather than $O(N)$).

```
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 $\lambda_{\textbf{obliv}}$ type system)

This program is also secure: no matter the value of s, the adversary learns nothing from the address trace. Consider Figure 2 which tabulates 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.

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

Fig. 2. Possible traces

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 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 $\Pr[\sigma_1; e_1 \longrightarrow^t \sigma_1'; e_1'] = q$ implies $\Pr[\sigma_2; e_2 \longrightarrow^t \sigma_2'; e_2'] = q$ and $\sigma_1'; e_1' \sim \sigma_2'; e_2'$.

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

The main contribution of this paper is $\lambda_{\textbf{obliv}}$, an expressive language whose type system guarantees that programs are probabilistically MTO. $\lambda_{\textbf{obliv}}$'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 bit$_\text{P}$ and bit$_\text{S}$ respectively, and coin flips are given type flip. Our formalism uses bits for simplicity; it is easy to generalize to (random fixed-width) integers, which is done in our implementation. Values of flip type are, like secret bits of type bit$_\text{S}$, invisible to the adversary. But a flip can be revealed by using cast$_\text{P}$ 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, $\lambda_{\mathbf{obliv}}$'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, $\lambda_{\mathbf{obliv}}$ relaxes the affinity constraint on flips passed to **castS**. In effect, programs can make many secret $\mathtt{bit}_S$ copies of a flip, 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. 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: $\Pr[\mathtt{sk}\ = 1] = \frac{3}{4}$ while $\Pr[\mathtt{sk}\ = 0] = \frac{1}{4}$. 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). The problematic correlation in Figure 3(b) is incorrectly allowed by ObliVM [Liu et al. 2015b], and is the root of its unsoundness.

Probability regions. $\lambda_{\mathbf{obliv}}$'s type system addresses the problem of probabilistic correlations leading to non-uniform distributions using a novel construct we call *probability regions*, which are static names that represent sets 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 programmers should write $\mathtt{flip}^{\rho}()$ for flipping a coin in region ρ, which then has type $\mathtt{flip}^{\rho}$. Bits derived from flips via **castS** carry the region of the original flip, so `bit` types also include a region ρ.

Regions form a partial order, and 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 $\rho' \sqsubset \rho$. 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. This kind of provable independence is a critical piece of our Tree ORAM implementation in Section 5.

3 FORMALISM

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

3.1 Syntax

Figure 4 shows the syntax for $\lambda_{\mathbf{obliv}}$. 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 $\mathtt{fun}_{y}(x{:}\tau).e$ where the function body may refer to itself using

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

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

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

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 $\texttt{bit}^{\perp}_{\ell}$) which are either $\texttt{O}$ or $\texttt{I}$ and annotated with their security label ℓ.[1] A security label ℓ is either S (secret) or P (public). Values with the label S are invisible to the adversary. Bit types include this security label along with a probability region ρ. The expression $\texttt{flip}^{\rho}()$ produces a flip value, i.e., a uniformly random bit of type $\texttt{flip}^{\rho}$. The annotation assigns the coin to region ρ. Coin flips are semantically secret, and have limited use; we can compute on one using $\texttt{mux}$ or $\texttt{xor}$, cast one to a public bit via $\texttt{cast}_{\mathrm{P}}$, or cast to a secret bit via $\texttt{cast}_{\mathrm{S}}$. To simplify the type system, casts only apply to values, however $\texttt{cast}_{\ell}(e)$ could be used as shorthand for $\texttt{let } x = e \texttt{ in } \texttt{cast}_{\ell}(x)$.

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

$\lambda_{\textbf{obliv}}$ 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.

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 $\texttt{bitv}_{\ell}(b)$, flip values $\texttt{flipv}(b)$, and reference locations $\texttt{locv}(\iota)$; 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 define evaluation contexts E (not shown) in the style of Felleisen and Hieb [1992] to enforce a left-to-right, call-by-value evaluation strategy.

[1] Bit literals are not values to create symmetry with the alternative, *mixed* semantics in the next section.

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

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

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

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

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

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

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

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

The semantics is defined using an abstract probability monad $\mathcal{M}$. Below the semantics we define the standard "denotational" discrete probability monad $\mathcal{D}$ [Giry 1982; Ramsey and Pfeffer 2002a]. The *standard* 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 the next section.

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 computable distributions [Huang and Morrisett 2016] or sub-probability distributions [Monniaux 2000]—we use the monad only to denote distributions of configurations which occur after a finite number of small-step transitions, which is total.

The definition of $\mathrm{step}_{\mathcal{M}}$ 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 $\mathtt{flip}^{\rho}()$ expression evaluates to either $\mathtt{flipv}(\mathtt{I})$ or $\mathtt{flipv}(\mathtt{O})$ as determined by $\mathrm{bit}(N)$, which for the monad $\mathcal{D}$ yields $1/2$ probability for each outcome. (The monad $\mathcal{D}$ does not use the N parameter in its definition of $\mathrm{bit}(N)$, but a later monad will.) The $\mathtt{cast}_\ell$ case converts a flip to a similarly-labeled bit value. The next few cases use the three-argument metafunction $\mathrm{cond}(b, X, Y)$, which returns X if b is $\mathtt{I}$, 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 $\mathtt{bitv}_\ell(\mathtt{I})$, and in reverse order when it is $\mathtt{bitv}_\ell(\mathtt{O})$. 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 flip, returning a flip.

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

Both $\mathrm{step}_{\mathcal{M}}$ and $\mathrm{nstep}_{\mathcal{M}}$ are *partial* in the usual way: They are undefined ("stuck") for nonsensical programs like $\mathtt{locv}(\iota)(\mathtt{bitv}_\ell(b))$ (treating a reference location as if it were a function). The $\lambda_{\mathbf{obliv}}$ type system, explained next, rejects such programs while also ensuring PMTO.

3.3 Type System

Figure 6 defines the type system for $\lambda_{\mathbf{obliv}}$ 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 $\bullet$, which are used to enforce affinity of flips. We discuss the three key features of the type system—affinity, probability regions, and information flow control—in turn.

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 $\mathtt{U}$ (freely duplicatable) or affine $\mathtt{A}$ (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 flips to bits via the $\mathtt{cast}_{\mathtt{S}}$ and $\mathtt{cast}_{\mathtt{P}}$ coercions, respectively. The first converts a $\mathtt{flip}^{\rho}$ to a $\mathtt{bit}^{\rho}_{\mathtt{S}}$ and does *not* make its argument inaccessible (it returns the original Γ) while the second converts to a $\mathtt{bit}^{\perp}_{\mathtt{P}}$ and does make it inaccessible (returning Γ'). 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 [Baker 1992].

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 a part that binds x and y and a part Γ that binds the rest; the Γ part is returned, dropping the x and y bindings. Both Let and Let-Tup similarly remove their bound variables.

$$\overset{\bullet}{\tau} \in \overset{\bullet}{\text{type}} ::= \tau \mid \bullet \quad (\textit{where } \tau \sqsubset \bullet) \qquad \Gamma \in \text{tcxt} \triangleq \mathit{var} \rightharpoonup \overset{\bullet}{\text{type}}$$

$$\kappa \in \text{kind} ::= \textsc{u} \mid \textsc{a} \quad (\textit{where } \textsc{u} \sqsubset \textsc{a}) \qquad (\Gamma_1 \sqcup \Gamma_2)(x) \triangleq \Gamma_1(x) \sqcup \Gamma_2(x)$$

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

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

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

$$\textsc{VarU}\ \frac{\begin{array}{c}\mathcal{K}(\Gamma(x)) = \textsc{u} \\ \Gamma(x) = \tau\end{array}}{\Gamma \vdash x : \tau \,;\, \Gamma} \qquad \textsc{VarA}\ \frac{\begin{array}{c}\mathcal{K}(\Gamma(x)) = \textsc{a} \\ \Gamma(x) = \tau\end{array}}{\Gamma \vdash x : \tau \,;\, \Gamma[x \mapsto \bullet]} \qquad \textsc{Bit}\ \frac{}{\Gamma \vdash b_{\ell} : \mathtt{bit}^{\perp}_{\ell} \,;\, \Gamma} \qquad \textsc{Flip}\ \frac{}{\Gamma \vdash \mathtt{flip}^{\rho}() : \mathtt{flip}^{\rho} \,;\, \Gamma}$$

$$\textsc{Cast-S}\ \frac{\Gamma \vdash x : \mathtt{flip}^{\rho} \,;\, \_}{\Gamma \vdash \mathtt{cast}_{\mathtt{S}}(x) : \mathtt{bit}^{\rho}_{\mathtt{S}} \,;\, \Gamma} \qquad \textsc{Cast-P}\ \frac{\Gamma \vdash x : \mathtt{flip}^{\rho} \,;\, \Gamma'}{\Gamma \vdash \mathtt{castp}(x) : \mathtt{bit}^{\perp}_{\mathtt{P}} \,;\, \Gamma'} \qquad \textsc{If}\ \frac{\begin{array}{c}\Gamma' \vdash e_1 : \tau \,;\, \Gamma''_1 \\ \Gamma \vdash e : \mathtt{bit}^{\perp}_{\mathtt{P}} \,;\, \Gamma' \quad \Gamma' \vdash e_2 : \tau \,;\, \Gamma''_2\end{array}}{\Gamma \vdash \mathtt{if}(e)\{e_1\}\{e_2\} : \tau \,;\, \Gamma''_1 \sqcup \Gamma''_2}$$

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

$$\textsc{Xor-Flip}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \mathtt{bit}^{\rho_1}_{\ell_1} \,;\, \Gamma' \\ \Gamma' \vdash e_2 : \mathtt{flip}^{\rho_2} \,;\, \Gamma'' \quad \rho_1 \sqsubset \rho_2\end{array}}{\Gamma \vdash \mathtt{xor}(e_1, e_2) : \mathtt{flip}^{\rho_2} \,;\, \Gamma''} \qquad \textsc{Ref}\ \frac{\Gamma \vdash e : \tau \,;\, \Gamma'}{\Gamma \vdash \mathtt{ref}(e) : \mathtt{ref}(\tau) \,;\, \Gamma'} \qquad \textsc{Read}\ \frac{\begin{array}{c}\mathcal{K}(\tau) = \textsc{u} \\ \Gamma \vdash e : \mathtt{ref}(\tau) \,;\, \Gamma'\end{array}}{\Gamma \vdash \mathtt{read}(e) : \tau \,;\, \Gamma'}$$

$$\textsc{Write}\ \frac{\Gamma \vdash e_1 : \mathtt{ref}(\tau) \,;\, \Gamma' \qquad \Gamma' \vdash e_2 : \tau \,;\, \Gamma''}{\Gamma \vdash \mathtt{write}(e_1, e_2) : \tau \,;\, \Gamma''} \qquad \textsc{Tup}\ \frac{\Gamma \vdash e_1 : \tau_1 \,;\, \Gamma' \qquad \Gamma' \vdash e_2 : \tau_2 \,;\, \Gamma''}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \,;\, \Gamma''}$$

$$\textsc{Fun}\ \frac{\begin{array}{c}\Gamma^{+} = \Gamma \uplus [x \mapsto \tau_1, y \mapsto (\tau_1 \to \tau_2)] \\ \Gamma^{+} \vdash e : \tau_2 \,;\, \Gamma^{+\prime} \quad \Gamma^{+\prime} = \Gamma \uplus [x \mapsto \_, y \mapsto \_]\end{array}}{\Gamma \vdash \mathtt{fun}_{y}(x : \tau_1).\ e : \tau_1 \to \tau_2 \,;\, \Gamma} \qquad \textsc{App}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \to \tau_2 \,;\, \Gamma' \\ \Gamma' \vdash e_2 : \tau_1 \,;\, \Gamma''\end{array}}{\Gamma \vdash e_1(e_2) : \tau_2 \,;\, \Gamma''}$$

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

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

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; if a variable is made inaccessible by one branch, it will be inaccessible in the joined environment. Contexts are joined pointwise, and the join of two pointed types $\overset{\bullet}{\tau}_1 \sqcup \overset{\bullet}{\tau}_2$ is $\bullet$ when either $\overset{\bullet}{\tau}_i$ is $\bullet$, the same as $\overset{\bullet}{\tau}_i$ when both $\overset{\bullet}{\tau}_i$ are equal and not $\bullet$, and undefined otherwise.

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 $\mathtt{bit}^{\rho}_{\ell}$ 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 $\mathtt{bit}^{\rho}_{\mathtt{S}}$) to a reference whose type is $\mathtt{ref}(\mathtt{bit}^{\rho}_{\mathtt{P}})$. Likewise, a function of type $\mathtt{bit}^{\rho}_{\mathtt{P}} \to \tau$ cannot be called with an argument of type $\mathtt{bit}^{\rho}_{\mathtt{S}}$, 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 more restrictive: rule If requires the guard to be public 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 be. The Mux-Flip rule always returns flips, which are invisible to the adversary, so the guard can be 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 $\perp$, as do public bits produced by `castP`, per rule Cast-P.

Regions form a join semi-lattice. The type system maintains the invariant that flips at region ρ are probabilistically independent of all secret bits in regions ρ' when strictly ordered $\rho' \sqsubset \rho$. Strict ordering is used because it is *irreflexive* and *asymmetric*. The semantic property of interest—probabilistic independence—is likewise irreflexive (except for point distributions), and asymmetry restricts future mux operations between values in one direction only; we say more below.

Consider the 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. 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 of its components.

Because freshly generated random bits are always independent of each other, the programmer is free to choose any regions when generating them via $\texttt{flip}^{\rho}()$ expressions. However, once chosen, the ordering establishes an invariant which constrains the order in which mux operations can occur subsequently in the program. Requiring strict region ordering for mux operations is enough to reject the example from the end of Section 2.4, as it could produce a non-uniform coin sk. We recast the example below, labeled (a), using regions $\rho_1 \sqsubset \rho_2$.

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

(a) Incorrect example

```
let sx = flip^ρ1 () in
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 $\texttt{bit}^{\rho_1}_{\mathsf{S}}$ 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., $\rho_1 \not\sqsubset \rho_1$. The program labeled (b) above is well-typed. Here, 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 by Mux-Flip. It is easy to see that both sy and sz are uniformly distributed and independent of sx.

Rule Xor-Flip permits xor'ing a secret with a flip, returning a flip, as long as the secret's region and the flip's region are well ordered, which preserves uniformity.

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

$$\boxed{\text{obs} \in (\text{exp} \rightarrow \dot{\text{exp}}) \times (\text{store} \rightarrow \dot{\text{store}}) \times (\text{config} \rightarrow \dot{\text{config}}) \times (\text{trace} \rightarrow \dot{\text{trace}})}$$

$$\begin{array}{ll}
\text{obs}(x) & \triangleq x \\
\text{obs}(\texttt{fun}_y(x : \tau).\ e) & \triangleq \texttt{fun}_y(x : \tau).\ \text{obs}(e) \\
\text{obs}(\texttt{bitv}_P(b)) & \triangleq \texttt{bitv}_P(b) \\
\text{obs}(\texttt{bitv}_S(b)) & \triangleq \bullet \\
\text{obs}(\texttt{flipv}(b)) & \triangleq \bullet \\
\text{obs}(\texttt{locv}(\iota)) & \triangleq \bullet \\
\text{obs}(b_P) & \triangleq b_P \\
\text{obs}(b_S) & \triangleq \bullet \\
\text{obs}(\texttt{flip}^\rho()) & \triangleq \texttt{flip}^\rho() \\
\text{obs}(\texttt{cast}_\ell(v)) & \triangleq \texttt{cast}_\ell(\text{obs}(v))
\end{array}
\qquad
\begin{array}{ll}
\text{obs}(\texttt{mux}(e_1, e_2, e_3)) & \triangleq \texttt{mux}(\text{obs}(e_1), \text{obs}(e_2), \text{obs}(e_3)) \\
\text{obs}(\texttt{xor}(e_1, e_2)) & \triangleq \texttt{xor}(\text{obs}(e_1), \text{obs}(e_2)) \\
\text{obs}(\texttt{if}(e_1)\{e_2\}\{e_3\}) & \triangleq \texttt{if}(\text{obs}(e_1))\{\text{obs}(e_2)\}\{\text{obs}(e_3)\} \\
\text{obs}(\texttt{ref}(e)) & \triangleq \texttt{ref}(\text{obs}(e)) \\
\text{obs}(\texttt{read}(e)) & \triangleq \texttt{read}(\text{obs}(e)) \\
\text{obs}(\texttt{write}(e_1, e_2)) & \triangleq \texttt{write}(\text{obs}(e_1), \text{obs}(e_2)) \\
\text{obs}(\langle e_1, e_2 \rangle) & \triangleq \langle \text{obs}(e_1), \text{obs}(e_2) \rangle \\
\text{obs}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & \triangleq \texttt{let}\ x = \text{obs}(e_1)\ \texttt{in}\ \text{obs}(e_2) \\
\text{obs}(\texttt{let}\ x, y = e_1\ \texttt{in}\ e_2) & \triangleq \texttt{let}\ x, y = \text{obs}(e_1)\ \texttt{in}\ \text{obs}(e_2) \\
\text{obs}(e_1(e_2)) & \triangleq \text{obs}(e_1)(\text{obs}(e_2))
\end{array}$$

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

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

Fig. 7. Adversary observability

We might be tempted not to order regions but instead 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 x and y are mutually independent. 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

The main metatheoretic result of this paper is that $\lambda_{\textbf{obliv}}$'s type system ensures probabilistic memory trace obliviousness (PMTO). This section defines this property, and then walks through its proof.

4.1 What is PMTO?

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

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.[2]

Proposition 4.1 (Probabilistic Memory Trace Obliviousness (PMTO)).
If: e_1 and e_2 are closed source expressions, $\vdash e_1 : \tau$, $\vdash e_2 : \tau$ and $\mathrm{obs}(e_1) = \mathrm{obs}(e_2)$
Then: (1) $\utilde{\mathrm{nstep}}_{\mathcal{D}}(N, \varnothing, e_1)$ and $\utilde{\mathrm{nstep}}_{\mathcal{D}}(N, \varnothing, e_2)$ are defined
And: (2) $\widetilde{\mathrm{obs}}(\utilde{\mathrm{nstep}}_{\mathcal{D}}(N, \varnothing, e_1)) = \widetilde{\mathrm{obs}}(\utilde{\mathrm{nstep}}_{\mathcal{D}}(N, \varnothing, e_2))$.

(1) ensures that information is not leaked due to lack of progress, i.e., if either program gets "stuck," and that the main property (2) applies to all related, well-typed source expressions e_1 and e_2.

4.2 Proof Approach

The remainder of this section works through our proof of PMTO (Theorem 4.7) which we complete in the following steps: (1) we develop a new probability monad called "intensional distributions" which simplifies reasoning about conditional independence between probabilistic values (§4.4); (2) we define an alternative syntax, semantics and type system for $\lambda_{\mathbf{obliv}}$ programs called the "mixed semantics" which uses intensional distributions to simplify inductive reasoning about the adversary's view of probabilistic secret values (§4.3, §4.5); (3) we show that evaluation in the mixed semantics corresponds exactly with the ground truth semantics through simulation lemmas; (4) we prove that key invariants about probabilistic values are ensured by well-typed mixed terms, and that terms remain well-typed throughout evaluation—this establishes PMTO for the mixed semantics; and (5) we demonstrate PMTO for the ground truth semantics as a consequence of lemmas established in steps (3–4) and a soundness lemma relating equivalent distributions of mixed terms to adversary-equivalent distributions of standard terms.

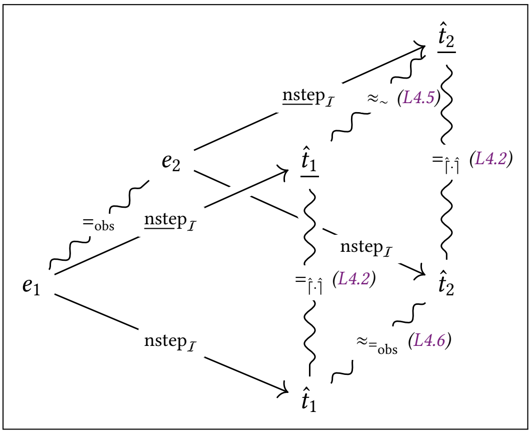
$\underline{\hat{t}_2}$
$\underline{\mathrm{nstep}}_{\mathcal{I}}$
$\approx_{\sim}$ (L4.5)
$=_{\hat{\lceil}\cdot\hat{\rceil}}$ (L4.2)
e_2
$\underline{\hat{t}_1}$
$=_{\mathrm{obs}}$
$\underline{\mathrm{nstep}}_{\mathcal{I}}$
$\mathrm{nstep}_{\mathcal{I}}$
e_1
$\hat{t}_2$
$=_{\hat{\lceil}\cdot\hat{\rceil}}$ (L4.2)
$\approx_{=_{\mathrm{obs}}}$ (L4.6)
$\mathrm{nstep}_{\mathcal{I}}$
$\hat{t}_1$

Fig. 8. Proof Approach as a Diagram

In Figure 8 we summarize the structure of this proof approach in a diagram. On the left are two programs e_1 and e_2 which are equal modulo adversary observation $=_{\mathrm{obs}}$, which translates to $\mathrm{obs}(e_1) = \mathrm{obs}(e_2)$ as sketched in Proposition 4.1, and means e_1 and e_2 agree on public values and program structure but may differ in secrets. The rightward moving arrows represent running each program in either the ground truth semantics $\mathrm{step}_{\mathcal{I}}$—the same semantics from Figure 5 but instantiated with the intensional distribution monad $\mathcal{I}$—and the mixed semantics $\underline{\mathrm{step}}_{\mathcal{I}}$. Each of these executions result in intensional distributions of standard and mixed traces, respectively. In step (3) above we prove Lemma 4.2 to show these distributions are equivalent according to $=_{\hat{\lceil}\cdot\hat{\rceil}}$ which uses $\hat{\lceil}\cdot\hat{\rceil}$ to project distributions of mixed traces to distributions of standard traces. In step (4) above we prove Lemma 4.5 to establish PMTO for the mixed semantics; i.e., that the resulting distributions of mixed traces are equivalent modulo an underlying low-equivalence relation $\approx_{\sim}$. In

[2] Noninterference properties are often stated with a non-empty store. Our notion of expression equivalence is simpler, and supports low-equivalent expressions that pre-populate such a store, so there is no loss of generality.

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

$$\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. 9. Mixed Language Semantics, where $\hat{b} \in \mathcal{I}(\mathbb{B})$ is a distributional bit value (see text)

step (5) we prove Lemma 4.6, which combines results from (3–4) to establish PMTO for the standard semantics (instantiated with $\mathcal{I}$)—the resulting distributions of standard traces are equivalent modulo equality of adversary observations, notated $\approx_{=_{\text{obs}}}$. The last step of PMTO (Theorem 4.7) is not shown: Lemma 4.3 proves via simulation that the intensional distribution monad $\mathcal{I}$ corresponds with the usual denotational probability monad presented in Section 3.

4.3 Mixed Semantics

An intuitive approach to proving Proposition 4.1 is to prove that a single-step version of it holds for $\text{step}_{\mathcal{D}}$, and then use that fact in an inductive proof over $\text{nstep}_{\mathcal{D}}$. Unfortunately, proving the single-step version quickly runs into trouble. Consider a source program $\texttt{cast}_P(\texttt{flip}^\rho())$ which steps to each of the expressions $\texttt{cast}_P(\texttt{flipv}(\texttt{I}))$ and $\texttt{cast}_P(\texttt{flipv}(\texttt{O}))$ with probability 1/2. These expressions are observationally equivalent—the adversary's view of each is $\texttt{cast}_P(\bullet)$. 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 $\texttt{bitv}_P(\texttt{I})$ and the second produces a point distribution of the expression $\texttt{bitv}_P(\texttt{O})$, 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 $\texttt{flipv}(\hat{b})$ where the $\hat{b}$ represents either $\texttt{I}$ or $\texttt{O}$ with equal probability. Doing this is like treating $\texttt{flip}^\rho()$ expressions lazily, and lines up (mixed) traces with the adversary's view $\bullet$.

The mixed semantics amends the syntax of $\texttt{flipv}$ and $\texttt{bitv}_\ell$ 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 values inside the pair to be correlated, we represent them using what we call *intensional distributions*—intensional distributions are written $\mathcal{I}(A)$ and discussed in the next subsection.

The mixed semantics is shown in Figure 9. The mixed semantics step function $\underline{\mathrm{step}}(N, \underline{\sigma}, \underline{e})$ maps a configuration, $\underline{\varsigma} \triangleq \underline{\sigma}, \underline{e}$ to an intensional distribution of configurations $\mathcal{I}(\underline{\mathrm{config}})$. 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 $\mathtt{flip}^{\rho}()$ and $\mathtt{cast}_{\ell}(\underline{v})$. For the first, the standard semantics samples from the fresh uniform distribution immediately, while the mixed semantics produces a single uniform distributional value. This distributional value is sampled at the evaluation of $\mathtt{cast}_{P}$, which matches the adversary's view.

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

4.4 Capturing Correlations with Intensional Distributions

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

Example. Consider the program $\mathtt{let}\ x = \mathtt{flip}^{\rho}()\ \mathtt{in}\ \langle \mathtt{castp}(x), \mathtt{castp}(x)\rangle$.[3]After two evaluation steps in the standard semantics, the program will be reduced to either $\langle \mathtt{castp}(\mathtt{flipv}(\mathtt{I})), \mathtt{castp}(\mathtt{flipv}(\mathtt{I}))\rangle$ or $\langle \mathtt{castp}(\mathtt{flipv}(\mathtt{O})), \mathtt{castp}(\mathtt{flipv}(\mathtt{O}))\rangle$, with equal probability. The standard rules for $\mathtt{castp}$ would then yield (equally likely) $\langle \mathtt{bitvp}(\mathtt{I}), \mathtt{bitvp}(\mathtt{I})\rangle$ and $\langle \mathtt{bitvp}(\mathtt{O}), \mathtt{bitvp}(\mathtt{O})\rangle$. In the mixed semantics this program will evaluate in two steps to $\langle \mathtt{castp}(\mathtt{flipv}(\hat{b})), \mathtt{castp}(\mathtt{flipv}(\hat{b}))\rangle$ where $\hat{b}$ is a distributional value. At this point, the mixed semantics rule for $\mathtt{castp}$ uses monadic bind to sample $\hat{b}$ to yield some b (which is either $\mathtt{I}$ or $\mathtt{O}$) and return it as a point distribution. The semantics needs to "remember" the bit chosen for the first $\mathtt{castp}$ 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. As shown in the upper left of Figure 10, an intensional distribution $\mathcal{I}(A)$ over a set A is a binary tree with elements a 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 ‹$\hat{x}_1$ $\hat{x}_2$› 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, ‹‹3 4› ‹3 5›› is an intensional distribution of numbers in a scenario where two coins have been flipped. There are four possible worlds. ‹3 4› is the world where the 0th coin came up heads. 3 is the outcome in the world where both coins came up heads, while 4 is the outcome where the 0th coin was heads but the 1th coin was tails. ‹3 5› is the world where the 0th coin came up tails, with 3 the outcome when the 1th coin was heads, and 5 when it was tails.

We can derive the probabilities of particular outcomes by counting the number of paths that reach them. 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

[3] 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.

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

$$
\begin{array}{ll}
\_[\_] & \in \mathcal{I}(A) \times \mathrm{rpath} \rightharpoonup A \\
a[p] & \triangleq a \\
\langle \hat{x}_1\ \hat{x}_2 \rangle[\textcircled{\tiny H} :: p] & \triangleq \hat{x}_1[p] \\
\langle \hat{x}_1\ \hat{x}_2 \rangle[\textcircled{\tiny T} :: p] & \triangleq \hat{x}_2[p] \\
\\
\mathrm{support} & \in \mathcal{I}(A) \longrightarrow \wp(A) \\
\mathrm{support}(\hat{x}) & \triangleq \{a \mid \hat{x}[p] = a\} \\
\\
\pi_1 & \in \mathcal{I}(A) \longrightarrow \mathcal{I}(A) \\
\pi_1(a) & \triangleq a \\
\pi_1(\langle \hat{x}_1\ \hat{x}_2 \rangle) & \triangleq \hat{x}_1 \\
\\
\pi_2 & \in \mathcal{I}(A) \longrightarrow \mathcal{I}(A) \\
\pi_2(a) & \triangleq a \\
\pi_2(\langle \hat{x}_1\ \hat{x}_2 \rangle) & \triangleq \hat{x}_2
\end{array}
$$

$$
\Pr\left[\overline{\hat{x} \doteq x} \,\middle|\, \overline{\hat{y} \doteq y}\right] \triangleq \frac{\Pr\left[\overline{\hat{x} \doteq x}, \overline{\hat{y} \doteq y}\right]}{\Pr\left[\overline{\hat{y} \doteq y}\right]}
$$

$$
\begin{array}{ll}
\mathrm{height} & \in \mathcal{I}(A) \longrightarrow \mathbb{N} \\
\mathrm{height}(a) & \triangleq 0 \\
\mathrm{height}(\langle \hat{x}_1\ \hat{x}_2 \rangle) & \triangleq 1 + \max(\mathrm{height}(\hat{x}_1), \mathrm{height}(\hat{x}_2)) \\
\\
\mathrm{length} & \in \mathrm{rpath} \longrightarrow \mathbb{B} \\
\mathrm{length}(\cdot) & \triangleq 0 \\
\mathrm{length}(\_ :: p) & \triangleq 1 + \mathrm{length}(p) \\
\\
\mathrm{bit} & \in \mathbb{N} \longrightarrow \mathcal{I}(\mathbb{B}) \\
\mathrm{bit}(0) & \triangleq \langle \texttt{I}\ \texttt{O} \rangle \\
\mathrm{bit}(N+1) & \triangleq \langle \mathrm{bit}(N)\ \mathrm{bit}(N) \rangle \\
\\
\mathrm{return} & \in A \longrightarrow \mathcal{I}(A) \\
\mathrm{return}(a) & \triangleq a \\
\\
\mathrm{bind} & \in \mathcal{I}(A) \times (A \longrightarrow \mathcal{I}(B)) \longrightarrow \mathcal{I}(B) \\
\mathrm{bind}(a, f) & \triangleq f(a) \\
\mathrm{bind}(\langle \hat{x}_1\ \hat{x}_2 \rangle, f) & \triangleq \langle \mathrm{bind}(\hat{x}_1, \pi_1 \circ f)\ \mathrm{bind}(\hat{x}_2, \pi_2 \circ f) \rangle
\end{array}
$$

$$
\Pr\left[\overline{\hat{x} \doteq x}\right] \triangleq \frac{\left|\{p \mid \mathrm{length}(p) = h, \overline{\hat{x}[p] = x}\}\right|}{2^h}
$$

$$
\textit{where } h \triangleq \max(\overline{\mathrm{height}(\hat{x})})
$$

Fig. 10. Intensional Distributions

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 $\langle\langle 3\ 3\rangle\ \langle 4\ 5\rangle\rangle$ ascribes outcomes 3, 4, and 5 the same probabilities as $\langle\langle 3\ 4\rangle\ \langle 3\ 5\rangle\rangle$, but represents the situation in which the we always get 3 when 0th coin flip is heads. An equivalent representation of $\langle\langle 3\ 3\rangle\ \langle 4\ 5\rangle\rangle$ is $\langle 3\ \langle 4\ 5\rangle\rangle$. Although the 3 only appears once, it is logically extended to the larger sub-tree $\langle 3\ 3\rangle$ for the purposes of counting. 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 to copy leaves that appear at shorter height. Trees are equal $=$ when they are syntactically equal modulo these extensions.

In the figure, a path p through the tree is a sequence of coin flip outcomes, either $\textcircled{\tiny H}$ or $\textcircled{\tiny T}$. The operation $\hat{x}[p]$ follows a path p through the tree $\hat{x}$ going left on $\textcircled{\tiny H}$ and right on $\textcircled{\tiny T}$. 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 intensional distribution $\hat{x}$ is shown at the bottom of the figure. As with the example above, it counts the number of paths that have outcome x, scaled by the total possible worlds. The probability of an event involving multiple 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 9. 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 $\hat{x}$, and $\mathrm{bind}(\hat{x}, f)[p] = f(\hat{x}[p])[p]$ for all paths p.

Example revisited. Reconsider the example $\texttt{let}\ x = \texttt{flip}^{\rho}()\ \texttt{in}\ \langle \texttt{castp}(x), \texttt{castp}(x) \rangle$. According to the mixed semantics starting with $N = 0$, $\texttt{flip}^{\rho}()$ evaluates to $\texttt{flipv}(\langle \texttt{I}\ \texttt{O}\rangle)$, which is then (as precipitated by $\underline{\mathrm{nstep}}$) substituted for x in the body of the $\texttt{let}$, producing $\langle \texttt{castp}(\texttt{flipv}(\langle \texttt{I}\ \texttt{O}\rangle)), \texttt{castp}(\texttt{flipv}(\langle \texttt{I}\ \texttt{O}\rangle)) \rangle$. Now we apply the context rule for $\underline{E}[\underline{e}]$ where $\underline{E}$ is $\langle [], \texttt{castp}(\texttt{flipv}(\langle \texttt{I}\ \texttt{O}\rangle)) \rangle$ and $\underline{e}$ is $\texttt{castp}(\texttt{flipv}(\langle \texttt{I}\ \texttt{O}\rangle))$.

The rule invokes $\underline{\mathrm{step}}$ on the latter, which performs $\mathrm{do}\ b \leftarrow \langle\texttt{I O}\rangle\ ;\ \mathrm{return}(\underline{\sigma}, \texttt{bitv}_P(\mathrm{return}(b)))$ per the rule for $\texttt{castp}$. Per the definitions of bind and return, this will return the intensional *distribution of configurations* $\langle(\underline{\sigma}, \texttt{bitv}_P(\texttt{I}))\ (\underline{\sigma}, \texttt{bitv}_P(\texttt{O}))\rangle$. Back to the context rule, its use of bind will re-package up these possibilities with $\underline{E}$:

$$\langle(\underline{\sigma}, \langle\texttt{bitv}_P(\texttt{I}), \texttt{castp}(\texttt{flipv}(\langle\texttt{I O}\rangle))\rangle)\ (\underline{\sigma}, \langle\texttt{bitv}_P(\texttt{O}), \texttt{castp}(\texttt{flipv}(\langle\texttt{I O}\rangle))\rangle)\rangle$$

In this distribution of configurations there are two worlds—the left configuration occurs when the 0th coin flip is heads, and right when it is tails. Inside of each of these configurations is a distributional value $\texttt{flipv}(\langle\texttt{I O}\rangle)$, 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 "unreachable" 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:

$$\langle(\underline{\sigma}, \langle\texttt{bitv}_P(\texttt{I}), \texttt{castp}(\texttt{flipv}(\langle\texttt{I}\ \bullet\rangle))\rangle)\ (\underline{\sigma}, \langle\texttt{bitv}_P(\texttt{O}), \texttt{castp}(\texttt{flipv}(\langle\bullet\ \texttt{O}\rangle))\rangle)\rangle$$

The next step of the computation will force the distributional value to be $\texttt{I}$ in the left branch and $\texttt{O}$ in the right branch. Here's how. First, the definition of $\underline{\mathrm{nstep}}$ is a bind on the above distribution of configurations with $\underline{\mathrm{step}}$ as the function f passed to bind. The definition of bind constructs a new distribution tree which calls $\underline{\mathrm{step}}$ on the left configuration, and then takes the left branch (π_1) of the tree that comes back, and likewise for the right configuration and the right branch that comes back (π_2). Here $\underline{\mathrm{step}}$ will invoke cast and context rules similarly as before, returning a two-element tree with $\texttt{bitv}_P(\texttt{I})$ on the left and $\texttt{bitv}_P(\texttt{O})$ on the right. These occurrences of π_1 and π_2 "pick" the left ($\texttt{I}$ case) and right ($\texttt{O}$ case), respectively, resulting in the final configuration $\langle(\underline{\sigma}, \langle\texttt{bitv}_P(\texttt{I}), \texttt{bitv}_P(\texttt{I})\rangle)\ (\underline{\sigma}, \langle\texttt{bitv}_P(\texttt{O}), \texttt{bitv}_P(\texttt{O})\rangle)\rangle$

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.

Projection is defined in Figure 11. 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:

$$\begin{aligned}&\hat{\lceil}\langle(\underline{\sigma}, \langle\texttt{bitv}_P(\texttt{I}), \texttt{castp}(\texttt{flipv}(\langle\texttt{I O}\rangle))\rangle)\ (\underline{\sigma}, \langle\texttt{bitv}_P(\texttt{O}), \texttt{castp}(\texttt{flipv}(\langle\texttt{I O}\rangle))\rangle)\rangle\hat{\rceil}\\ &= \langle(\underline{\sigma}, \langle\texttt{bitv}_P(\texttt{I}), \texttt{castp}(\texttt{flipv}(\texttt{I}))\rangle)\ (\underline{\sigma}, \langle\texttt{bitv}_P(\texttt{O}), \texttt{castp}(\texttt{flipv}(\texttt{O}))\rangle)\rangle\end{aligned}$$

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

We prove that the projected, mixed semantics simulates the standard semantics.

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

To relate to "ground truth", we also prove that the standard semantics using intensional distributions $\mathcal{I}$ simulates the standard semantics using the denotational probability monad $\mathcal{D}$.

Lemma 4.3 (Simulation (Intensional)). $\Pr\left[\mathrm{nstep}_{\mathcal{I}}(N, \varnothing, e) \doteq t\right] = \Pr\left[\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e) \doteq t\right]$.

4.5 Mixed Semantics Typing

Our type system aims to ensure that $\texttt{cast}_P$ will produce $\texttt{I}$ and $\texttt{O}$ with equal probability, meaning neither outcome leaks information. We establish this invariant in the PMTO proof as a consequence of type preservation for mixed terms. The mixed term typing judgment extends typing of source-program expressions (Figure 6) with 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 12. Here, Σ is a *store context*, which maps store locations to types; it is used to type the store $\underline{\sigma}$ in rules Store-Cons

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

$$\begin{array}{llll}
\lceil x \rceil & \triangleq \mathrm{return}(x) & \lceil \mathtt{fun}_y(x : \tau).\ \underline{e} \rceil & \triangleq \mathrm{do}\ e \leftarrow \lceil \underline{e} \rceil \ ;\ \mathrm{return}(\mathtt{fun}_y(x : \tau).\ e) \\
\lceil \mathtt{locv}(\iota) \rceil & \triangleq \mathrm{return}(\mathtt{locv}(\iota)) & \lceil \mathtt{bitv}_\ell(\hat{b}) \rceil & \triangleq \mathrm{do}\ b \leftarrow \hat{b}\ ;\ \mathrm{return}(\mathtt{bitv}_\ell(b)) \\
\lceil b_\ell \rceil & \triangleq \mathrm{return}(b_\ell) & \lceil \mathtt{flipv}(\hat{b}) \rceil & \triangleq \mathrm{do}\ b \leftarrow \hat{b}\ ;\ \mathrm{return}(\mathtt{flipv}(b)) \\
\lceil \mathtt{flip}^\rho() \rceil & \triangleq \mathrm{return}(\mathtt{flip}^\rho()) & \lceil \mathtt{cast}_\ell(\underline{v}) \rceil & \triangleq \mathrm{do}\ v \leftarrow \lceil \underline{v} \rceil \ ;\ \mathrm{return}(\mathtt{cast}_\ell(v))
\end{array}$$

$$\begin{array}{ll}
\lceil \mathtt{mux}(\underline{e}_1, \underline{e}_2, \underline{e}_3) \rceil & \triangleq \mathrm{do}\ e_1 \leftarrow \lceil \underline{e}_1 \rceil \ ;\ e_2 \leftarrow \lceil \underline{e}_2 \rceil \ ;\ e_3 \leftarrow \lceil \underline{e}_3 \rceil \ ;\ \mathrm{return}(\mathtt{mux}(e_1, e_2, e_3)) \\
\lceil \mathtt{xor}(\underline{e}_1, \underline{e}_2) \rceil & \triangleq \mathrm{do}\ e_1 \leftarrow \lceil \underline{e}_1 \rceil \ ;\ e_2 \leftarrow \lceil \underline{e}_2 \rceil \ ;\ \mathrm{return}(\mathtt{xor}(e_1, e_2)) \\
\lceil \mathtt{if}(\underline{e}_1)\{\underline{e}_2\}\{\underline{e}_3\} \rceil & \triangleq \mathrm{do}\ e_1 \leftarrow \lceil \underline{e}_1 \rceil \ ;\ e_2 \leftarrow \lceil \underline{e}_2 \rceil \ ;\ e_3 \leftarrow \lceil \underline{e}_3 \rceil \ ;\ \mathrm{return}(\mathtt{if}(e_1)\{e_2\}\{e_3\}) \\
\lceil \mathtt{ref}(\underline{e}_1) \rceil & \triangleq \mathrm{do}\ e_1 \leftarrow \lceil \underline{e}_1 \rceil \ ;\ \mathrm{return}(\mathtt{ref}(e_1)) \\
\lceil \mathtt{read}(\underline{e}_1) \rceil & \triangleq \mathrm{do}\ e_1 \leftarrow \lceil \underline{e}_1 \rceil \ ;\ \mathrm{return}(\mathtt{read}(e_1)) \\
\lceil \mathtt{write}(\underline{e}_1, \underline{e}_2) \rceil & \triangleq \mathrm{do}\ e_1 \leftarrow \lceil \underline{e}_1 \rceil \ ;\ e_2 \leftarrow \lceil \underline{e}_2 \rceil \ ;\ \mathrm{return}(\mathtt{write}(e_1, e_2)) \\
\lceil \langle \underline{e}_1, \underline{e}_2 \rangle \rceil & \triangleq \mathrm{do}\ e_1 \leftarrow \lceil \underline{e}_1 \rceil \ ;\ e_2 \leftarrow \lceil \underline{e}_2 \rceil \ ;\ \mathrm{return}(\langle e_1, e_2 \rangle) \\
\lceil \mathtt{let}\ x = \underline{e}_1\ \mathtt{in}\ \underline{e}_2 \rceil & \triangleq \mathrm{do}\ e_1 \leftarrow \lceil \underline{e}_1 \rceil \ ;\ e_2 \leftarrow \lceil \underline{e}_2 \rceil \ ;\ \mathrm{return}(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \\
\lceil \mathtt{let}\ x, y = \underline{e}_1\ \mathtt{in}\ \underline{e}_2 \rceil & \triangleq \mathrm{do}\ e_1 \leftarrow \lceil \underline{e}_1 \rceil \ ;\ e_2 \leftarrow \lceil \underline{e}_2 \rceil \ ;\ \mathrm{return}(\mathtt{let}\ x, y = e_1\ \mathtt{in}\ e_2) \\
\lceil \underline{e}_1(\underline{e}_2) \rceil & \triangleq \mathrm{do}\ e_1 \leftarrow \lceil \underline{e}_1 \rceil \ ;\ e_2 \leftarrow \lceil \underline{e}_2 \rceil \ ;\ \mathrm{return}(e_1(e_2))
\end{array}$$

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

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

$$\hat{\lceil} \hat{\underline{t}} \hat{\rceil} \triangleq \mathrm{do}\ \underline{t} \leftarrow \hat{\underline{t}}\ ;\ \lceil \underline{t} \rceil \qquad \boxed{\hat{\lceil} \_ \hat{\rceil} \in \mathcal{I}(\underline{\mathrm{trace}}) \rightarrow \mathcal{I}(\mathrm{trace})}$$

Fig. 11. Mixed Semantics Projection

$$\Psi^F \in \mathrm{flipset} \triangleq \wp(\mathcal{I}(\mathbb{B})) \qquad \Psi^B \in \mathrm{bitset} \triangleq R \rightarrow \wp(\mathcal{I}(\mathbb{B})) \qquad \Psi \in \mathrm{fbset} ::= \Psi^F, \Psi^B \qquad \Phi \in \mathrm{history} ::= \overline{\hat{\underline{\varsigma}} \doteq \underline{\varsigma}}$$

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

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

Flip-Value

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

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

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

$\ldots$

Ref

$$\frac{\Psi, \Phi, \Sigma, \Gamma \vdash \underline{e} : \tau \ ;\ \Gamma', \Psi'}{\Psi, \Phi, \Sigma, \Gamma \vdash \mathtt{ref}(\underline{e}) : \mathtt{ref}(\tau) \ ;\ \Gamma', \Psi'}$$

Tup

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

$\ldots$

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. 12. Mixed Semantics Typing

and LocV as usual. Φ represents *trace history* which encodes the exact sequence of evaluation steps taken to reach the present one. The type system reasons about the probability of distributional values conditioned on this trace history having occurred. The Ψ is an *fbset*, which is a technical device used to collect all 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 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 evaluation context and store.

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 type 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 similarly handles the store and the expression. The rules combine two fbsets using the $\uplus$ operator. Per the top of the figure, it acts as disjoint union for flipsets but normal union for bitsets, mirroring the handling of affine and universal variables.

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 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 invariants are preserved.

Lemma 4.4 (Type Preservation). *If e is a closed source expression, $\underline{t}\cdot\underline{\varsigma} \in \text{support}(\underline{\text{nstep}}(N, \varnothing, e))$ and $\vdash e : \tau$, then there exists Σ and Ψ s.t. $\Phi, \Sigma \vdash \underline{\varsigma} : \tau, \Psi$ where $\Phi \triangleq \left[\underline{\text{nstep}}(N, \varnothing, e) \doteq \underline{t}\cdot\underline{\varsigma}\right]$.*

When a configuration takes any number of steps, the resulting configuration is well-typed under new trace history Φ. Updating Φ is not arbitrary—it is necessary to satisfy a proof obligation as used in a later lemma (PMTO (Mixed)). 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 proof of preservation uses a sublemma which shows typesafe substitution; this lemma makes crucial use of affinity to ensure that aggregated $\Psi_1 \uplus \Psi_2$ in contexts for compound expressions (e.g., pairs) are truly disjoint, which will be true only because the substitution is guaranteed to only occur in Ψ_1, Ψ_2, or neither, but not both.

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 trace history, and it is not true that uniformity conditioned on the current history Φ automatically implies uniformity in the new history Φ'; this must be proved. Because the second premise establishes independence from all other flips, we are able reestablish the first premise via the second after some other flip is revealed to complete the proof.

Note that we also prove a progress lemma to ensure that no well-typed evaluation reaches a stuck state; along with preservation, this lemma establishes standard type soundness for $\lambda_{\mathbf{obliv}}$ under the mixed semantics.

4.6 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 $\underline{e}_1$ and $\underline{e}_2$ are closed source expressions, $\vdash \underline{e}_1 : \tau$, $\vdash \underline{e}_2 : \tau$ and $\underline{e}_1 \sim \underline{e}_2$, then (1) $\underline{\mathrm{nstep}}(N, \varnothing, \underline{e}_1)$ and $\underline{\mathrm{nstep}}(N, \varnothing, \underline{e}_2)$ are defined, and (2) $\underline{\mathrm{nstep}}(N, \varnothing, \underline{e}_1) \approx_{\sim} \underline{\mathrm{nstep}}(N, \varnothing, \underline{e}_2)$.*

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 supplemental report [Darais et al. 2019]) and we can easily prove that it is implied by $\mathrm{obs}(e_1) = \mathrm{obs}(e_2)$ for source expressions. Mixed PMTO establishes equivalence of the distributions of mixed configurations modulo low-equivalence. We define two distributions as equivalent modulo an underlying equivalence relation as follows:

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

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 PMTO (Mixed) by induction over steps N and then unfolding the monadic definition of $\underline{\mathrm{nstep}}(N+1)$. The induction appeals to a single-step PMTO sublemma. (As mentioned in Section 4.3, such a proof would not have been possible in the standard semantics.) To use this one-step PMTO sublemma, it must be that the configuration at N steps is well-typed w.r.t. current trace history Φ; we get this well-typing w.r.t. Φ from Type Preservation, discussed earlier.

A final major lemma in our PMTO proof is a notion of soundness for low-equivalence on mixed terms, in particular, that 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 $\widehat{\mathrm{obs}}(\hat{\lceil} \underline{\hat{t}_1} \hat{\rceil}) \approx_{=} \widehat{\mathrm{obs}}(\hat{\lceil} \underline{\hat{t}_2} \hat{\rceil})$.*

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

We now complete the full proof of PMTO. The general strategy is to first consider two well-typed source programs which are equal modulo adversary observation. Next, these programs are transported to the mixed language, where low-equivalence is established. 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.

Theorem 4.7 (PMTO).
If: e_1 *and* e_2 *are closed source expressions,* $\vdash e_1 : \tau$, $\vdash e_2 : \tau$ *and* $\mathrm{obs}(e_1) = \mathrm{obs}(e_2)$
Then: (1) $\utilde{\mathrm{nstep}}_{\mathcal{D}}(N, \varnothing, e_1)$ *and* $\utilde{\mathrm{nstep}}_{\mathcal{D}}(N, \varnothing, e_2)$ *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 (see supplemental report). (2) is by the following:

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

□

A detailed proof is given in the supplemental report [Darais et al. 2019].

5 IMPLEMENTATION AND TREE-BASED ORAM CASE STUDY

We have implemented an interpreter and type checker for a language that extends $\lambda_{\mathbf{obliv}}$ in several (straightforward) ways. First, we add natural number literals and random values; these can be encoded in $\lambda_{\mathbf{obliv}}$ as fixed-width tuples of `bitv` and `flipv` respectively. We write them annotated with a security level, e.g., `2 S` or `2 P`, and write **rnd** `R ()` to generate a random number at region `R`. We write `natS` to be the type of a secret number in region ⊥; `natP` for the type of a public number; `R natS` for the type of a secret number in the region `R`. We also write `R` **rnd** to be the type of a random natural number in the region `R`. Second, we add arrays; in our code examples, we write `a[n]` and `a[n] ← e` to read and write array elements. An array of length N can be encoded in $\lambda_{\mathbf{obliv}}$ as an N-tuple of references, using nested conditional expressions to access the correct (public) index and swapping out affine contents, as must be done with references. Finally, we add records, which are like tuples but permit field accessor notation, `r.x`; if `x` is affine, doing so only consumes the field `x` rather than consuming all of `r`.

To demonstrate the expressiveness of $\lambda_{\mathbf{obliv}}$, we have used our extended language to program (and type check) a series of interesting oblivious algorithms. Section 5.2 presents a modern *non-recursive, tree-based ORAM* (NORAM), which is a key component of state-of-the-art ORAM implementations [Shi et al. 2011; Stefanov et al. 2013; Wang et al. 2015]. To our knowledge, ours is the first implementation automatically verified to be oblivious. Building on this NORAM, Section 5.3 presents a full *recursive* ORAM. Type checking it requires some advanced (but standard) language features we have not implemented, including region polymorphism, recursive and variant types, and existential quantification. Finally, the supplemental report [Darais et al. 2019] presents a mostly complete implementation of *oblivious stacks* (ostacks), a kind of oblivious data structure [Wang et al. 2014] that builds on top of NORAM. The $\lambda_{\mathbf{obliv}}$ type system is not powerful enough to reason that ostacks' use of NORAM is safe; the region ordering requirement is too strong. Sections 6 and 7 discuss integrating $\lambda_{\mathbf{obliv}}$'s type system with a general-purpose logic as a way to potentially overcome this limitation. Our type checker and all the examples are online at https://github.com/plum-umd/oblivml.

5.1 Tree-based ORAM: Overview

A complete ORAM implements the same API as a standard array: A `read` operation takes an ORAM `oram` and index `i` as arguments, and returns data `d` stored at that index; a `write` operation updates `oram` at `i` with a given `d`. We assume that the ORAM contents and the indexes are not visible to the adversary (i.e., they are encrypted). A simple implementation is a *Trivial ORAM.* It consists of an array of N "buckets," each of which consists of an index i and data d. A read at index j iterates over the entire array and retrieves the data associated with j, if present. The data is returned when the iteration is complete (or a default value is returned, if j is not present). Since each read touches every bucket, nothing is leaked about i. Of course, this is very inefficient—the read takes time $O(N)$ where N is the size of the array. (The code example in Figure 1(b) does something similar.)

A tree-based ORAM [Shi et al. 2011; Stefanov et al. 2013; Wang et al. 2015] offers better performance. It breaks its implementation into two parts. The first is a tree-like structure noram for storing the actual data blocks; this is called a *non-recursive ORAM* (or NORAM) for reasons that will be clear in the next subsection. The second part is the *position map* pm that maps logical data block indexes to *position tags* that indicate the block's position in the tree.

NORAMs do not implement read and write operations directly; instead they implement two more-primitive operations called noram_readAndRemove (or noram_rr, for short) and noram_add. The former reads the designated data block from noram and also removes it, while the latter adds the given data. Putting it all together, a Tree ORAM read from index i works in four steps: (1) retrieve tag t from pm[i]; (2) call noram_rr noram i t to remove the data d at i using t to assist the lookup; (3) update pm[i] with a randomly generated tag t2; and (4) call noram_add noram i t2 d to add back data d, but with the new tag, before returning it. An ORAM write has the same four steps, but in step (4) we add the provided data, rather than the original. (A fifth step in both cases, *eviction*, will be explained later.) As with the example in Figure 1(c), non-recursive ORAM combines randomness (and its tree structure) to avoid having $O(N)$ cost for the entire map: Under the right assumptions, these operations take time $O(\log(N))$.

The position tags mask the relationship between a logical index and the location of its corresponding data block in the tree. As blocks are read and written, they are shuffled around in the tree, and their new locations are recorded in the position map. As such, two ORAM read operations to the same index i will involve different access patterns in a way that leaks nothing about the index *assuming* lookups and updates to the position map itself leak no information. This assumption could be satisfied by making the position map a Trivial ORAM, but then we would lose our performance benefits. In the next subsection we simply assume we have a leak-free position map and in Section 5.3 we show how one can be obtained by efficiently storing the position map *recursively* in the NORAM tree structure itself.

5.2 Tree-based Non-recursive ORAM

Now we present the details of our implementation of tree-based NORAM in $\lambda_{\textbf{obliv}}$.

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

```
type block  = { is_dummy : R bitS ; idx : R natS ; tag : R natS ; data : (R ∨ R' rnd) * (R ∨ R' rnd) }
type bucket = block array
type noram  = bucket array
```

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. In the operations discussed below, all functions prefixed with trivial are operations over buckets.

The region R ∨ R' should be read as "R join R'" and corresponds to the join operation, ⊔, over regions ρ in Section 3. Notice that we have R ⊏ R ∨ R', which will be important when discussing well-typedness of **mux** in the discussion that follows. We choose type (R ∨ R' **rnd**) * (R ∨ 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. The code for noram_rr is given below; we explain it just afterward.

```
1  let rec trivial_rr_h (troram : bucket) (idx : R natS) (i : natP) (acc : block) : block =
2    if i = length(troram) then acc
3    else
4      (* read out the current block, replace with dummy *)
5      let curr = bucket[i] ← (dummy_block ()) in
6      (* check if the current block is non−dummy, and its index matches the queried one *)
7      let swap : R bits = !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 : bucket) (idx : R natS) : (R ∨ R' rnd) * (R ∨ 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 : noram) (idx : R natS) (tag : natP) (level : natP) (acc : block) : 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 : noram) (idx : R natS) (tag : natP) : (R ∨ R' rnd) * (R ∨ R' rnd) =
29   let ret = noram_rr_h noram idx tag 0 (dummy_block ()) in
30   ret.data
```

noram_rr takes the NORAM noram and the index idx of the desired element as arguments. 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 tag's type natP means it is publicly visible. Initially it is stored, secretly, in the position map, but prior to passing it to this function it must be revealed (via **castP**) because it (or derivatives of it) will be used to index the arrays that make up the NORAM, and array indexes are always adversary-visible.

noram_rr works by calling noram_rr_h which recursively works its way down the identified path. It maintains an accumulator, acc : 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 ∨ R' **rnd**) * (R ∨ 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 is an array. As such, we can read from noram without writing a new value (line 24). However, the bucket type has contents which are kind A, since the type of its contents are tuples which contain type R ∨ R' **rnd**. So, when we index into members of values of type 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 : noram → (idx : R natS) → (tag : R natS) → (data : (R ∨ R' rnd) * (R ∨ 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 root bucket's 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, our NORAM employs an additional eviction routine. It is called after both noram_add and noram_rr, to move blocks closer to the leaf buckets. This routine maintains the key invariant that each data block should 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. The simple eviction strategy we implement (due to Shi et al. [2011]) 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.

5.3 Recursive ORAM

As described in Section 5.1, a complete ORAM combines a non-recursive ORAM with a position map. So far, we have not said where the position map should be stored, and how. One approach is to implement it as just a regular array stored in hidden memory, e.g., on-chip (invisible to the adversary) in a secure processor deployment of ORAM (see Section 2.1). However, this is not possible for MPC-based deployments, in which both parties secret-share the map, and thus the adversary can observe the access pattern on the map itself. To block this side channel, we could implement the position map itself as an ORAM, e.g., a Trivial ORAM. But to do so would ruin the efficiency gain of our tree-based NORAM, since the position map lookup would have time $O(N)$, as compared to $O(\log(N))$ time for noram_rr and noram_add.

We could implement the position map in a NORAM in an attempt to get back logarithmic-time efficiency, but doing so seems to "kick the can down the road" because we now need another position map for our position map! We can close this cycle by having each recursively defined position map be smaller than the previous. In particular, to implement a map with N integer keys we can use a map of N/c keys, each of which maps to c values, for a small constant c. Lookup of key k translates to looking up key k/c in the smaller map, and then returning the $(k\%c)$th value (which takes time c to do obliviously). We can apply this idea recursively, ultimately yielding $\log_c(N)$ maps numbered $i = 1 \ldots \log_c(N)$, where map i has $\frac{N}{c^i}$ keys (and each key maps to c values). We can implement each map at level i as a NORAM until i is large enough that we can use a Trivial ORAM to tie it off (e.g., when $\frac{N}{c^i}$ is 4). The complexity of looking up a key will thus be $\sum_{i=1}^{\log_c(N)} O(\log(\frac{N}{c^i}) + c)$. Setting c to be a constant 2 means that the complexity of the lookup procedure is $O(\log(N)^2)$. This construction is called a *recursive ORAM*.

Data Definition and Operations. A recursive ORAM thus has the type oram, given below.

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

The data blocks are stored in the noram at index 0 in the first component, an noram array; the remaining norams in that array consist of progressively smaller position maps, finally ending in a trival ORAM, the second component (a bucket).

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 (initially, 0).

```
1   let rec tree_rr_h (oram : oram) (idx : natS) ( level : natP): (R ∨ R' rnd) * (R ∨ R' rnd) =
2     let (norams, troram) = oram in
3     let levels : natP = length(norams) in
4     if level >= levels then trivial_rr troram idx
5     else
6       let (r0, r1) : (R ∨ R' rnd) * (R ∨ R' rnd) = tree_rr_h oram (idx / 2) ( level + 1) in
7       let (r0 ', tag) = mux(idx % 2 = 0, rnd (R ∨ R') (), r0) in
8       let (r1 ', tag) = mux(idx % 2 = 1, tag, r1) in
9       let _ = tree_add_h oram (idx / 2) ( level + 1) (r0 ', r1 ') in
10      noram_rr norams[level] idx (castP tag)
11
12  let tree_rr (oram : oram) (idx : natS): (R ∨ R' rnd) * (R ∨ R' rnd) =
13    tree_rr_h oram idx 0
```

In the code above, the level indicates the embedded NORAM from which to read. For example, when level is 0, the data NORAM should be read. For any other level > 0, the NORAM will be one of the embedded position maps. Recall that each NORAM at level i has its position map at level $i + 1$, with the exception of the very last NORAM which uses a Trivial ORAM for its position map. The recursive call to tree_rr_h on line 6 reads out of the next level's map, returning the pair (r0, r1). These are the two possible position tags for nrorams[level]—we should return r0 if idx % 2 = 0 and r1 if idx % 2 = 1. The muxes on lines 7 and 8 obliviously achieve this, reading the proper result into tag, replacing it with a freshly generated tag, to satisfy the affinity requirement. Line 9 writes the updated block (r0 ', r1 ') for idx / 2 back, using an analogous tree_add_h routine, for which a level can be specified. Finally, line 10 reveals the retrieved position tag for index idx, so that it can be passed to noram_rr. Since level 0 corresponds to the actual data of the ORAM, that is what will finally be 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 : R ∨ R' rnd = rnd R ∨ R' () in
2  let sec_tag = castS new_tag in (* does NOT consume new_tag *)
3  let (r0, r1) : (R ∨ R' rnd) * (R ∨ R' rnd) = tree_rr_h 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_h 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_h 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.

We note that neither tree_rr nor tree_add are complete ORAM operations on their own: to implement a full ORAM read, for example, we would need to call tree_rr with a call to tree_add.

Discussion. Unfortunately (as astute readers may have noticed), the code snippet for add will not type check. In particular, the sec_tag argument has type R ∨ R' natS but noram_add requires it to have type R natS. 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 to have a strictly smaller region than its data (i.e., R ⊏ R ∨ R').

We can solve this problem by extending the language to support variant and recursive types, existential quantification, and *region polymorphism*, where region-polymorphic variables may have ordering constraints. With these changes, the type of oram would be the following:

```
type (R1,R2) block = { is_dummy : R1 bitS ; idx : R1 natS ; tag : R1 natS ; data: (R2 rnd) * (R2 rnd) }
      where R1 ⊏ R2
type (R1,R2) bucket = (R1,R2) block array
type (R1,R2) noram = (R1,R2) bucket array
 type (R1, R2) oram =
   Trivial of (R1,R2) bucket
 | Recursive of ∃R. (R, R1) noram * (R1,R2) oram where R1 ⊏ R2
```

We re-present the definitions for the elements of noram, which we now parameterize with polymorphic region variables. For block, we add the constraint that R1 ⊏ R2. When originally presenting NORAM, this wasn't needed because we were using concrete regions—notice that R and R ∨ R' from our previous noram definition satisfy the constraint on R1 and R2, respectively, in the new definition. Type oram is also parameterized by region variables, and is now a recursive variant: it can be either a trivial ORAM or a recursive ORAM. The latter is an NORAM paired with an ORAM, which acts as its position map. Importantly, the region R2 of the ORAM data is properly ordered with the region of the position map R1. The code would be roughly the same as the code given above, except that rather than indexing the norams array at each recursive level, it simply recurses down the oram datastructure. Constructing such a datastructure would require satisfying the region constraints at each level, which is easy to do by simply using distinct regions for each region variable. Along with our other code examples at https://github.com/plum-umd/oblivml, we show how this could work using OCaml-style functors.

Oblivious Stacks. Other oblivious data structures [Wang et al. 2014] can be built in $\lambda_{\textbf{obliv}}$, and on top of noram in particular. The supplemental report presents a development of probabilistic oblivious stacks (*ostacks*). As explained there, the strict ordering of probability regions imposes a similar problem on ostacks as on recursive ORAMs, but for ostacks the problem cannot be addressed with straightforward language extensions. Instead, different reasoning principles are required. It's possible these can be integrated into $\lambda_{\textbf{obliv}}$ via inclusion of a general-purpose logic.

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; Chan et al. 2019; 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 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, 2014] developed a type system that ensures programs are MTO, generalizing a line of prior works on (language-enforced) timing channel security [Agat 2000], program counter security [Molnar et al. 2006]. In Liu et al's work, 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. $\lambda_{\mathbf{obliv}}$'s key advance is that it applies to *probabilistic* programs. It need not assume the existence of ORAM as a primitive; rather, $\lambda_{\mathbf{obliv}}$'s probabilistic nature is sufficient to allow us to *program* ORAM, per Section 5. Thus we can express state-of-the-art algorithmic results and formally reason about the security of their implementations, building a bridge between algorithmic and programming language techniques.

ObliVM [Liu et al. 2015b] is a language for programming probabilistically 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, 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.

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; Zhang et al. 2019b], essentially aims to bound possible leakage; by contrast, we enforce that *no* information leaks due to a program's execution.

Our intensional distributions—while a novel syntactic device instrumental to our proof approach—are readily interpretable as measurable sets over infinite streams of bits, and there is prior work which has considered such models such as Kozen's seminal treatment [Kozen 1979] among others [Barker 2016; Huang and Morrisett 2016; Park et al. 2008; Ramsey and Pfeffer 2002b; Ścibior et al. 2015]. A novelty in our model is support for conditional probabilistic reasoning. This reasoning is enabled by our interpretation of monadic bind as conditioning on outcomes, and performing sampling of new bits via operations external to monad operations; doing so is in contrast to prior work which interprets monadic bind directly as (effectively) sampling new random bits.

There is a rich history for *reasoning* about probabilistic programs [Sato et al. 2019], in particular relational properties [Barthe et al. 2014, 2017b; Hsu 2017] and program logics [Barthe et al. 2018; Rand and Zdancewic 2015], including trace properties [Smith et al. 2019], privacy properties [Barthe et al. 2015; Gaboardi et al. 2013; Reed and Pierce 2010], obliviousness properties [Ohrimenko et al. 2016], and uniformity and independence [Barthe et al. 2017a]. Much of this work is focused on verification techniques for some program of interest, and not on proof techniques for establishing metatheoric properties of entire languages (e.g., via a type system).

Perhaps the most closely related program logic to our setting is Probabilistic Separation Logic (PSL) [Barthe et al. 2020]. PSL is a variant of separation logic in which separating conjunction models probabilistic independence. It supports reasoning about (conditional) independence and uniformity, which are both also key ideas in $\lambda_{\textbf{obliv}}$. There is a similar connection between some of PSL's proof rules and $\lambda_{\textbf{obliv}}$'s type rules; e.g., $\lambda_{\textbf{obliv}}$'s Mux-Flip rule and PSL's RCond rule both reason about conditional independence. It would be interesting to explore how to embed $\lambda_{\textbf{obliv}}$'s type system in PSL's logic, which might simplify reasoning about security for PSL, and open up reasoning about correctness for $\lambda_{\textbf{obliv}}$ programs. It might also permit proofs of uniformity that $\lambda_{\textbf{obliv}}$'s strict region ordering currently forbid. How to combine these two is not obvious, though, as PSL works on an imperative "while" language with a fixed set of (global) variables, while $\lambda_{\textbf{obliv}}$ is functional, and supports dynamically-sized data structures. Interesting future work!

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. To the best of our knowledge, by type checking an implementation of tree-based ORAM in $\lambda_{\textbf{obliv}}$ we have carried out the first automated proof that this algorithm is secure.

While $\lambda_{\textbf{obliv}}$ advances the state of the art in security type systems, there are still oblivious algorithms it is not powerful enough to check. As noted at the end of Section 5 (and the supplemental report), the strict ordering on probability regions is sound but cannot handle some idioms. More precise reasoning about probabilities is needed. We believe that a promising way forward is to integrate $\lambda_{\textbf{obliv}}$'s type-level mechanisms with richer systems for formal reasoning. For example, we could adopt the approach of *semantic typing*, embedding $\lambda_{\textbf{obliv}}$'s type rules as lemmas in a richer logic, as done in RustBelt [Jung et al. 2018] or Fuzzi [Zhang et al. 2019b]. The logic of Barthe et al. [2020] is a good candidate, but it needs further extensions too. Another benefit of embedding $\lambda_{\textbf{obliv}}$'s type system into a full logic is that we can use the logic to reason about algorithm correctness, something $\lambda_{\textbf{obliv}}$ does not do.

ACKNOWLEDGMENTS

We thank Aseem Rastogi, Kesha Heitala, Joe Near, and the anonymous reviewers for comments on earlier drafts of this paper, and Elaine Shi for helpful discussions throughout our process. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1563722, CNS-1314857, CNS-1111599 and CCF-1901278; by DARPA under contracts FA8750-15-2-0104 and FA8750-16-C-0022; and by ODNI/IARPA via 2019-1902070008. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF, DARPA, ODNI, IARPA, or the U.S. Government.

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: "Look Ma, No Garbage!";. *SIGPLAN Not.* 27, 8 (Aug. 1992), 89–98. https://doi.org/10.1145/142137.142162

Tyler Barker. 2016. A Monad for Randomized Algorithms. *Electronic Notes in Theoretical Computer Science* 325 (2016), 47 – 62. https://doi.org/10.1016/j.entcs.2016.09.031 The Thirty-second Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXII).

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, Thomas Espitau, Benjamin Gr\'egoire, Justin Hsu, and Pierre-Yves Strub. 2017a. Proving uniformity and independence by self-composition and coupling. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing)*, Thomas Eiter and David Sands (Eds.), Vol. 46. EasyChair, 385–403. https://doi.org/10.29007/vz48

Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. 2014. Probabilistic Relational Verification for Cryptographic Implementations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 193–205. https://doi.org/10.1145/2535838.2535847

Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 55–68.

Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2017b. Coupling Proofs Are Probabilistic Product Programs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 161–174. https://doi.org/10.1145/3009837.3009896

Gilles Barthe, Justin Hsu, and Kevin Liao. 2020. A Probabilistic Separation Logic. *PACMPL* 4, POPL (2020).

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*.

T-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. 2019. Foundations of Differentially Oblivious Algorithms. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '19)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2448–2467. http://dl.acm.org/citation.cfm?id=3310435.3310585

David Darais, Ian Sweet, Chang Liu, and Michael Hicks. 2019. *A Language for Probabilistically Oblivious Computation*. Technical Report abs/1711.09305. CoRR. arXiv:1711.09305

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*.

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. 2013. Linear dependent types for differential privacy. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 357–370.

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.

Justin Hsu. 2017. Probabilistic Couplings for Probabilistic Reasoning. *CoRR* abs/1710.09951 (2017). arXiv:1710.09951 http://arxiv.org/abs/1710.09951

Daniel Huang and Greg Morrisett. 2016. An Application of Computable Distributions to the Semantics of Probabilistic Programming Languages. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–363.

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)*.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* POPL (2018).

Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*.

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*.

Dexter Kozen. 1979. Semantics of Probabilistic Programs. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science (SFCS '79)*. IEEE Computer Society, Washington, DC, USA, 101–114. https://doi.org/10.1109/SFCS.1979.38

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

Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*.

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*.

David Monniaux. 2000. Abstract Interpretation of Probabilistic Semantics. In *Seventh International Static Analysis Symposium (SAS'00) (Lecture Notes in Computer Science)*. Springer Verlag, 322–339. https://doi.org/10.1007/978-3-540-45099-3_17

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).

oblivm-www 2019. ObliVM Open Source Release. www.oblivm.com.

Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-party Machine Learning on Trusted Processors. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*. USENIX Association, Berkeley, CA, USA, 619–636. http://dl.acm.org/citation.cfm?id=3241094.3241143

Sungwoo Park, Frank Pfenning, and Sebastian Thrun. 2008. A Probabilistic Language Based on Sampling Functions. *ACM Trans. Program. Lang. Syst.* 31, 1, Article 4 (Dec. 2008), 46 pages. https://doi.org/10.1145/1452044.1452048

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

Norman Ramsey and Avi Pfeffer. 2002b. Stochastic Lambda Calculus and Monads of Probability Distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, New York, NY, USA, 154–165. https://doi.org/10.1145/503272.503288

Robert Rand and Steve Zdancewic. 2015. VPHL. *Electron. Notes Theor. Comput. Sci.* 319, C (Dec. 2015), 351–367. https://doi.org/10.1016/j.entcs.2015.12.021

Jason Reed and Benjamin C Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. *ACM Sigplan Notices* 45, 9 (2010), 157–168.

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*.

Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. 2019. Formal Verification of Higher-order Probabilistic Programs: Reasoning About Approximation, Convergence, Bayesian Inference, and Optimization. *Proc. ACM Program. Lang.* 3, POPL, Article 38 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290351

Adam Ścibior, Zoubin Ghahramani, and Andrew D. Gordon. 2015. Practical Probabilistic Programming with Monads. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. ACM, New York, NY, USA, 165–176. https://doi.org/10.1145/2804302.2804317

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

Calvin Smith, Justin Hsu, and Aws Albarghouthi. 2019. Trace Abstraction Modulo Probability. *Proc. ACM Program. Lang.* 3, POPL, Article 39 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290352

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).

Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-order Execution. In *USENIX Security*.

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*.

Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C. Pierce, and Aaron Roth. 2019a. Fuzzi: A Three-level Logic for Differential Privacy. *PACMPL* 3, ICFP (2019).

Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C. Pierce, and Aaron Roth. 2019b. Fuzzi: A Three-Level Logic for Differential Privacy. *CoRR* abs/1905.12594 (2019). arXiv:1905.12594 http://arxiv.org/abs/1905.12594

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).