A Language for Probabilistically Oblivious Computation

DAVID DARAIS, University of Vermont
IAN SWEET, University of Maryland
CHANG LIU, University of California, Berkeley
MICHAEL HICKS, University of Maryland

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

1 INTRODUCTION

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

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

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

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

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

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

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

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

$\lambda_{\textbf{obliv}}$'s type system, presented in Section 3, ensures that revelations are uniformly distributed by treating randomly generated numbers as *affine*, meaning they cannot be freely copied. This prohibition prevents revealing the same number twice, which would be problematic if allowed because the second revelation is not uniformly distributed. Unfortunately, strict affinity is too strong for implementing oblivious algorithms, which of which require the ability to make copies of random numbers which are later revealed. $\lambda_{\textbf{obliv}}$'s type system addresses this by allowing random numbers to be copied as non-affine secret values which can never be revealed. Moreover, $\lambda_{\textbf{obliv}}$ enforces that random numbers do not influence the choice of whether or not they are revealed, since this could also result in a non-uniform revelation. For example, a $\lambda_{\textbf{obliv}}$ program cannot copy a random number to a secret, look at that secret, and then decide whether or not to reveal the original random number. The type system prevents this problem by using a new mechanism we call *probability regions* to track the probabilistic (in)dependence of values in the program. (Probability regions are missing in ObliVM, and their absence is the source of ObliVM's unsoundness.) Section 4.7 outlines proof that $\lambda_{\textbf{obliv}}$ enjoys PMTO by relating its semantics to a novel *mixed semantics* whose terms operate on distributions directly, which makes it easier to state and prove the PMTO property. We have mechanized a significant portion of the metatheory in Agda as supplemental material.

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

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

caveats. As far as we are aware, our implementations constitute the first automated "proofs" that these data structures are indeed oblivious. Though further improvements can be made, $\lambda_{\textbf{obliv}}$'s support for randomness strictly generalizes prior work on oblivious by typing, and thus constitutes a promising step forward. (Section 6 discusses related work.)

2 OVERVIEW

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

2.1 Threat Model

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

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

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

2.2 Oblivious Execution

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

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

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

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

(a) Leaky program

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

(b) Deterministic MTO program

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

(c) Probabilistic MTO program

Fig. 1. Code examples

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. We can illustrate this approach with the program in Figure 1(c), which hints at the basic approach to implementing an ORAM. Rather than deterministically store s0 and s1 in positions 0 and 1 of B, respectively, the program scrambles their locations according to a coin flip, sk, generated by the call to **flip**, and not visible to the adversary. Using the **mux** on line 2, if sk is 0 then s0 and s1 will be copied to s0' and s1', respectively, but if sk is 1 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. The coercion **castP** is a no-op that makes plain that the value s' is made visible to the adversary.

In terms of memory accesses, this program is more efficient: It is reading B only once, rather than twice. One can argue that more work is done overall, but as we will see in Section 5, this basic idea does scale up to build full ORAMs with access times of $O(log_c\ N)$ for some c. This program is also secure: no matter the value of s, the adversary can learn nothing from the address trace. To see why, consider the Figure 2 which categorizes the four possible traces (the memory indexes used to access B) depending on the possible values of s and sk. This table makes plain that our program is not *deterministically* MTO. Looking at column sk=0, we can see that a program that has s=0 may produce trace 0,1,0 while a program that uses s=1 may produce trace 0,1,1; MTO programs may not produce different traces when using different secrets.

But this is not actually a problem. Assuming that sk = 0 and sk = 1 are equally likely, we can see that address traces 0,1,0 and 0,1,1 are also equally likely no matter whether s = 0 or s = 1. More specifically, if we assume the adversary's expectation for secret values is uniformly distributed, then after *conditioning* on knowledge of the third memory access, the adversary's expectation for the secret remains unchanged, and thus nothing is learned about s. This probabilistic model of knowledge and learning by the adversary 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 $\Pr[\sigma_1; e_1 \longrightarrow^t \sigma_1'; e_1'] = q_1$ and $\Pr[\sigma_2; e_2 \longrightarrow^t \sigma_2'; e_2'] = q_2$ implies $q_1 = q_2$ and $\sigma_1'; e_1' \sim \sigma_2'; e_2'$ when $q_i > 0$.

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

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

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

Fig. 2. Possible traces

$\lambda_{\textbf{obliv}}$'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 observed by the adversary—even if the value interacted with secrets, like the secret memory layout of an ORAM—it is always uniformly distributed, which is a sufficient condition for obliviousness.

Affinity. In $\lambda_{\textbf{obliv}}$, public and secret bits are given types $\textbf{bit}_{\textbf{P}}$ and $\textbf{bit}_{\textbf{S}}$ respectively. Coin flips are given type **flip** and are produced by executing **flip**(). Values of **flip** type are, like secret bits of type $\textbf{bit}_{\textbf{S}}$, invisible to the adversary. Coin flips can be copied to secret bits with the **castS** coercion and can be revealed to public bits with the **castP** coercion. Line 2 of Figure 1(c) has an example of the former, and line 8 is an example of the latter. We use bits for simplicity; it is easy to generalize to integers (as done in our implementation).

$\lambda_{\textbf{obliv}}$'s type system enforces that a coin flip can be made public at most once. It does this by treating values of type **flip** affinely: they cannot be duplicated and they are consumed by operations other than **castS**. For example, in Figure 1(c), variable sk is not consumed on line 2 when passed to **castS**, but is consumed when passed as the second argument to the **xor** on line 7. Likewise, s' has type **flip** and it is consumed when its cast-to-public result is used as an index on line 8.

Why is affinity important? Consider the following example:

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

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. Treating **flip** values affinely solves this problem: the appearance of sx on line 2 consumes that variable, so it cannot be used again on the problematic line 4.

Probability regions. Unfortunately, affinity alone is insufficient. This is because, for reasons of needed expressiveness (e.g., to implement Tree ORAMs, as shown in Section 5), **castS** can convert coin flips to non-affine secret bits, and then use these to potentially affect the distribution of coin flips whose eventual public visibility could depend on a secret. To see how, consider the following code:

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

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).

$\lambda_{\textbf{obliv}}$'s type system addresses this problem with a novel construct we call *probability regions*. Both bit and **flip** types are ascribed a region set R containing one or more *static* probability regions ρ. Each static region ρ represents the set of *dynamic* coin flips generated by **flip** () instructions annotated with ρ (reminiscent of a points-to location in alias analysis [Emami et al. 1994]). We have elided the region name so far, but normally should write $\textbf{flip}^{\rho}()$ for flipping a coin in static region ρ, which then has type $\textbf{flip}^{\{\rho\}}$. We should also write $\textbf{bit}_{\mathrm{S}}^{R}$ and $\textbf{bit}_{\mathrm{P}}^{R}$ for bit types; we leave off R when it is $\varnothing$. Regions allow the type system to reason that random values are probabilistically independent when static region sets R_1 and R_2 do not overlap. If the static regions sets overlap, then it is possible that there is a probabilistic dependence between values. Probabilistic dependence may inadvertently cause a previously uniform distribution to become non-uniform, which would then be problematic if revealed to the adversary. For example, a sufficient condition for the bitwise xor operation $b_1 \oplus b_2$ to return a uniform result is when b_2 is both uniform *and* probabilistically independent of b_1. These conditions successfully rule out xor of a bit with itself $b \oplus b$ which is always 0 (*i.e.* not uniform) even if b is uniform. In our type system, we encode the assumption and guarantee of uniformity via the special $\textbf{flip}^{R}$ type (as opposed to $\textbf{bit}_{\ell}^{R}$), and the assumption of independence via non-overlapping static probability region sets.

We can see regions at work in the problematic example above: the region set of the secret bit **castS**(sx) is the same region set as sx, since **castS**(sx) was derived from sx. As such, the two region sets overlap, and therefore the values may not be independent. This lack of independence is problematic when conditioning on **castS**(sx) to return sx for similar reasons as described for xor above, and means that the output is no longer guaranteed to be uniform. On the other hand, if the guard of a **mux** has a region set that is *independent* of the region sets of its branches, then the uniformity of the output is not threatened. Bits that never were (or were influenced by) random values have region set $\varnothing$ because they are independent of any random value. As such, s and sk when used on line 7 of Figure 1(c) are independent.

The next two sections formalize these ideas in a core calculus and describe our proof that $\lambda_{\textbf{obliv}}$'s type system is sufficient to ensure PMTO.

3 FORMALISM

This section presents the syntax, sampling semantics, and type system of $\lambda_{\textbf{obliv}}$, and formally states the PMTO security property it enjoys.

3.1 Syntax

Figure 3 shows the syntax for $\lambda_{\textbf{obliv}}$. The term language is expressions e. These include values v, which comprise variables x, pairs $\langle v, v\rangle$ (of type $\tau \times \tau$), and recursive functions $\textbf{fun}_{y}(x : \tau).\ e$ (of type $\tau \to \tau$). A function's body e may refer to itself using variable y.

$$
\begin{array}{rrll}
\ell \in \mathsf{label} & ::= & \mathsf{P} \mid \mathsf{S} & \text{public and secret} \\
& & (\textit{where}\ \mathsf{P} \sqsubset \mathsf{S}) & \text{security labels} \\
\rho \in \mathsf{rvar} & ::= & \ldots & \text{probability region} \\
R \in \mathsf{rexp} & \triangleq & \wp(\mathsf{rvar}) & \text{region sets} \\
b \in \mathbb{B} & ::= & \mathbf{O} \mid \mathbf{I} & \text{bits} \\
x, y \in \mathsf{var} & ::= & \ldots & \text{variables} \\
v \in \mathsf{val} & ::= & x & \text{variable values} \\
& \mid & \langle v, v \rangle & \text{tuple values} \\
& \mid & \mathbf{fun}_y(x{:}\tau).e & \text{function values} \\
\tau \in \mathsf{type} & ::= & \mathbf{bit}_\ell^R & \text{non-random bit} \\
& \mid & \mathbf{flip}^R & \text{secret uniform bit} \\
& \mid & \tau \times \tau & \text{tuple} \\
& \mid & \tau \to \tau & \text{function}
\end{array}
$$

$$
\begin{array}{rrll}
e & \in & \mathsf{exp} & \\
e & ::= & v & \text{value expressions} \\
& \mid & b_\ell & \text{bit literal} \\
& \mid & \mathbf{flip}^{\{\rho\}\cup R}() & \text{coin flip in } \rho \\
& \mid & \mathbf{cast}_\ell(e) & \text{cast flip to bit} \\
& \mid & \mathbf{mux}(e,e,e) & \text{atomic conditional} \\
& \mid & \mathbf{xor}(e,e) & \text{bit xor} \\
& \mid & \mathbf{if}(e)\{e\}\{e\} & \text{branch conditional} \\
& \mid & \langle e, e \rangle & \text{tuple creation} \\
& \mid & \mathbf{let}\ x = e\ \mathbf{in}\ e & \text{variable binding} \\
& \mid & \mathbf{let}\ x, y = e\ \mathbf{in}\ e & \text{tuple elimination} \\
& \mid & e(e) & \text{fun. application}
\end{array}
$$

Fig. 3. $\lambda_{\mathbf{obliv}}$ Syntax

Expressions also include bit literals b_ℓ (of type $\mathbf{bit}_\ell^\varnothing$) which are either $\mathbf{O}$ or $\mathbf{I}$ and annotated with their security label ℓ.[2] A security label ℓ is either S (secret) or P (public). Values with the former label are invisible to the adversary. Bit types include this security label along with a static probability region set R. The expression $\mathbf{flip}^{\{\rho\}\cup R}()$ produces a coin flip, i.e., a randomly generated bit of type $\mathbf{flip}^{\{\rho\}\cup R}$. The annotation assigns the coin to region ρ operationally (per Section 3.4) but then weakens it to a larger set $\{\rho\} \cup R$; we write $\mathbf{flip}^\rho()$ as a shorthand when R is $\varnothing$. Coin flips are semantically secret, and have limited use; we can compute on one using $\mathbf{mux}$ or $\mathbf{xor}$ (since doing so preserves the uniformity of the distribution in the presence of specific probabilistic independence conditions), cast one to a public bit via $\mathbf{cast}_\mathsf{P}(e)$, or cast to a secret bit via $\mathbf{cast}_\mathsf{S}(e)$.

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

$\lambda_{\mathbf{obliv}}$ 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. Notable omissions in our language are support for arrays and polymorphic types, which are useful when implementing ORAMs. We omit them here to simplify the presentation; they pose no specific technical hurdle but do add notational comlpexity. Our prototype interpreter has both arrays and region polymorphism (Section 5) and our mechanization of $\lambda_{\mathbf{obliv}}$ in Agda (included as supplemental material) is done for a language with ML-style reference to demonstrate correctness of the proof approach in the presence of arrays.

3.2 Sampling Semantics

Figure 4 shows a small-step operational semantics for $\lambda_{\mathbf{obliv}}$ programs. The main judgment has form $e \rightsquigarrow_q e'$ which states that a expression e steps to expression e' with probability q (a rational number). The top of the figure contains some new and extended syntax. Values (and, by extension,

[2]Bit literals are not values in order to distinguish them from those that appear at runtime. We do this to align the syntax with an alternative semantics developed later for the purposes of proving probabilistic MTO. In the alternative semantics, runtime bit values carry much more information than just a bit and label.

$$
\begin{array}{llll}
v \in \mathsf{val} ::= \ldots & \text{extended}\ldots & e \in \mathsf{exp} ::= \ldots & \text{extended}\ldots \\
\quad \mid \mathbf{bitv}_\ell(b) & \text{bit value} & E \in \mathsf{context} ::= \ldots & \text{extended}\ldots \\
\quad \mid \mathbf{flipv}(b) & \text{uniform bit value} & t \in \mathsf{trace} ::= \epsilon \mid t \cdot e & \text{traces}
\end{array}
$$

$$
\begin{array}{llll}
b_\ell & \rightsquigarrow_1 \ \mathbf{bitv}_\ell(b) & \mathbf{if}(\mathbf{bitv}_\mathsf{P}(\mathbf{I}))\{e_1\}\{e_2\} & \rightsquigarrow_1 \ e_1 \\
\mathbf{flip}^{\{\rho\}\cup R}() & \rightsquigarrow_{1/2} \ \mathbf{flipv}(\mathbf{I}) & \mathbf{if}(\mathbf{bitv}_\mathsf{P}(\mathbf{O}))\{e_1\}\{e_2\} & \rightsquigarrow_1 \ e_2 \\
\mathbf{flip}^{\{\rho\}\cup R}() & \rightsquigarrow_{1/2} \ \mathbf{flipv}(\mathbf{O}) & \mathbf{let}\ x = v\ \mathbf{in}\ e & \rightsquigarrow_1 \ e[v/x] \\
\mathbf{cast}_\mathsf{S}(\mathbf{flipv}(b)) & \rightsquigarrow_1 \ \mathbf{bitv}_\mathsf{S}(b) & \mathbf{let}\ x_1, x_2 = \langle v_1, v_2 \rangle\ \mathbf{in}\ e & \rightsquigarrow_1 \ e[v_1/x_1][v_2/x_2] \\
\mathbf{cast}_\mathsf{P}(\mathbf{flipv}(b)) & \rightsquigarrow_1 \ \mathbf{bitv}_\mathsf{P}(b) & (\underbrace{\mathbf{fun}_y(x : \tau).\ e}_{v_1})(v_2) & \rightsquigarrow_1 \ e[v_1/y][v_2/x]
\end{array}
\qquad \boxed{e \rightsquigarrow_q e}
$$

$$
\begin{array}{ll}
\mathbf{mux}(\mathbf{bitv}_{\ell_1}(\mathbf{I}), \mathbf{bitv}_{\ell_2}(b_1), \mathbf{bitv}_{\ell_3}(b_2)) & \rightsquigarrow_1 \ \langle \mathbf{bitv}_{\ell_1 \sqcup \ell_2 \sqcup \ell_3}(b_1), \mathbf{bitv}_{\ell_1 \sqcup \ell_2 \sqcup \ell_3}(b_2) \rangle \\
\mathbf{mux}(\mathbf{bitv}_{\ell_1}(\mathbf{O}), \mathbf{bitv}_{\ell_2}(b_1), \mathbf{bitv}_{\ell_3}(b_2)) & \rightsquigarrow_1 \ \langle \mathbf{bitv}_{\ell_1 \sqcup \ell_2 \sqcup \ell_3}(b_2), \mathbf{bitv}_{\ell_1 \sqcup \ell_2 \sqcup \ell_3}(b_1) \rangle \\
\mathbf{mux}(\mathbf{bitv}_{\ell_1}(\mathbf{I}), \mathbf{flipv}(b_1), \mathbf{flipv}(b_2)) & \rightsquigarrow_1 \ \langle \mathbf{flipv}(b_1), \mathbf{flipv}(b_2) \rangle \\
\mathbf{mux}(\mathbf{bitv}_{\ell_1}(\mathbf{O}), \mathbf{flipv}(b_1), \mathbf{flipv}(b_2)) & \rightsquigarrow_1 \ \langle \mathbf{flipv}(b_2), \mathbf{flipv}(b_1) \rangle \\
\mathbf{xor}(\mathbf{bitv}_{\ell_1}(b_1), \mathbf{flipv}(b_2)) & \rightsquigarrow_1 \ \mathbf{flipv}(b_1 \oplus b_2) \\
E[e] & \rightsquigarrow_q \ E[e'] \ \textit{where}\ e \rightsquigarrow_q e'
\end{array}
$$

$$
\widetilde{x} \in \mathcal{D}(X) \triangleq X \to \{q \in \mathbb{Q} \mid 0 \le q \le 1\} \ \textit{where} \sum_{x \in X} \widetilde{x}(x) = 1 \qquad \boxed{\widetilde{x} \in \mathcal{D}(X)}
$$

$$
\Pr\left[\widetilde{x} \stackrel{?}{=} x\right] \triangleq \widetilde{x}(x)
$$

$$
\mathcal{S}(t \cdot e) \triangleq \lambda(t' \cdot e').\ \begin{cases} q & \textit{if}\ e \rightsquigarrow_q e'\ \textit{and}\ t' = t \cdot e \\ 1 & \textit{if}\ e \in \mathsf{val}\ \textit{and}\ t' \cdot e' = t \cdot e \\ 0 & \textit{otherwise} \end{cases} \qquad \boxed{\mathcal{S} \in \mathsf{trace} \to \mathcal{D}(\mathsf{trace})}
$$

$$
\mathcal{S}^*(\widetilde{t}) \triangleq \lambda t'.\ \sum_t \Pr\left[\widetilde{t} \stackrel{?}{=} t\right] \Pr\left[\mathcal{S}(t) \stackrel{?}{=} t'\right] \qquad \boxed{\mathcal{S}^* \in \mathcal{D}(\mathsf{trace}) \to \mathcal{D}(\mathsf{trace})}
$$

$$
\mathcal{S}^*_{n+1}(\widetilde{t}) \triangleq \mathcal{S}^*_n(\mathcal{S}^*(\widetilde{t})) \qquad \mathcal{S}^*_0(\widetilde{t}) \triangleq \widetilde{t} \qquad \boxed{\mathcal{S}^*_n \in \mathcal{D}(\mathsf{trace}) \to \mathcal{D}(\mathsf{trace})}
$$

Fig. 4. $\lambda_{\mathbf{obliv}}$ Sampling Semantics

expressions) are extended with forms for (non-random) bits $\mathbf{bitv}_\ell(b)$, and generated coins $\mathbf{flipv}(b)$; these do not appear in source programs. We use Hieb-Felleisen context-based semantics [Felleisen and Hieb 1992], defining contexts E (not shown) for a left-to-right, call-by-value evaluation strategy.

The rules let binding, pair deconstruction, and function application are standard, and use a substitution-based semantics. The rules for **if** are also standard; these branch on the value forms for public bits. Literals b_ℓ evaluate in one step to bit values.[3]

There are two rules for $\mathbf{flip}^{\{\rho\}\cup R}()$ expressions, one each for evaluating to $\mathbf{flipv}(\mathbf{I})$ or $\mathbf{flipv}(\mathbf{O})$ with probability $1/2$. All of the other rules have probability 1, aside from the context rule, whose probability q is inherited from the reduction of its substituted-in expression e.

The $\mathbf{cast}_\mathsf{S}$ and $\mathbf{cast}_\mathsf{P}$ rules, respectively, convert a $\mathbf{flipv}(b)$ to either a secret or public bit.

The rules for **mux** return the second two arguments of the **mux** in order when the first argument is $\mathbf{bitv}_\ell(\mathbf{I})$, and in reverse order when it is $\mathbf{bitv}_\ell(\mathbf{O})$. When those two arguments are bits, the security label is updated with the join of the labels of all elements in involved. This is not needed for coins, since these are always secret. The rule for **xor** permits xor-ing a bit with a coin, returning a coin.

An execution trace t is a sequence of expressions $\epsilon \cdot e_1 \cdot \ldots \cdot e_N$ with each e_i and e_{i+1} in the sequence justified by proof of judgment $e_i \rightsquigarrow_q e_{i+1}$. For purposes of proving PMTO we are interested in the distribution of possible traces resulting from executing a program. We define a distribution $\widetilde{x} \in \mathcal{D}(X)$ as a function from X to a rational number q which sums to 1. We write

[3]We make bit literals b_ℓ distinct from runtime bit values $\mathbf{bitv}_\ell(b)$ to mirror the *mixed* semantics developed in Section 4.7.

$$\dot{\tau} \in \dot{\text{type}} ::= \tau \mid \bullet \ (\textit{where } \tau \sqsubset \bullet) \qquad \Gamma \in \text{tcxt} \triangleq \mathit{var} \rightharpoonup \dot{\text{type}}$$
$$\kappa \in \text{kind} ::= \mathbf{U} \mid \mathbf{A} \ (\textit{where } \mathbf{U} \sqsubset \mathbf{A}) \qquad \Gamma_1 \sqcup \Gamma_2 \triangleq \lambda x.\ \Gamma_1(x) \sqcup \Gamma_2(x)$$

$$\mathcal{K}(\mathbf{bit}^R_\ell) \triangleq \mathcal{K}(\tau_1 \to \tau_2) \triangleq \mathbf{U} \qquad \mathcal{K}(\mathbf{flip}^R) \triangleq \mathbf{A} \qquad \mathcal{K}(\tau_1 \times \tau_2) \triangleq \mathcal{K}(\tau_1) \sqcup \mathcal{K}(\tau_2) \qquad \boxed{\mathcal{K} \in \text{type} \to \text{kind}}$$

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

$$\textsc{Bit}\ \frac{}{\Gamma \vdash b_\ell : \mathbf{bit}^{\varnothing}_\ell \ ; \Gamma} \qquad \textsc{Flip}\ \frac{}{\Gamma \vdash \mathbf{flip}^{\{\rho\}\cup R}() : \mathbf{flip}^{\{\rho\}\cup R} \ ; \Gamma} \qquad \textsc{Cast-S}\ \frac{\Gamma \vdash v : \mathbf{flip}^R \ ; \_}{\Gamma \vdash \mathbf{cast}_\mathrm{S}(v) : \mathbf{bit}^R_\mathrm{S} \ ; \Gamma} \qquad \textsc{Cast-P}\ \frac{\Gamma \vdash e : \mathbf{flip}^R \ ; \Gamma'}{\Gamma \vdash \mathbf{cast}_\mathrm{P}(e) : \mathbf{bit}^{\varnothing}_\mathrm{P} \ ; \Gamma'}$$

$$\textsc{VarU}\ \frac{\mathcal{K}(\Gamma(x)) = \mathbf{U} \quad \Gamma(x) = \tau}{\Gamma \vdash x : \tau \ ; \Gamma} \qquad \textsc{VarA}\ \frac{\mathcal{K}(\Gamma(x)) = \mathbf{A} \quad \Gamma(x) = \tau}{\Gamma \vdash x : \tau \ ; \Gamma[x \mapsto \bullet]}$$

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

$$\textsc{Mux-Bit}\ \frac{\Gamma \vdash e_1 : \mathbf{bit}^{R_1}_{\ell_1} \ ; \Gamma' \quad \Gamma' \vdash e_2 : \mathbf{bit}^{R_2}_{\ell_2} \ ; \Gamma'' \quad \ell = \ell_1 \sqcup \ell_2 \sqcup \ell_3 \quad \Gamma'' \vdash e_3 : \mathbf{bit}^{R_3}_{\ell_3} \ ; \Gamma''' \quad R = R_1 \cup R_2 \cup R_3}{\Gamma \vdash \mathbf{mux}(e_1, e_2, e_3) : \mathbf{bit}^R_\ell \times \mathbf{bit}^R_\ell \ ; \Gamma'''}$$

$$\textsc{Mux-Flip}\ \frac{\Gamma \vdash e_1 : \mathbf{bit}^{R_1}_{\ell_1} \ ; \Gamma' \quad R_1 \perp\!\!\!\perp R_2 \quad \Gamma' \vdash e_2 : \mathbf{flip}^{R_2} \ ; \Gamma'' \quad R_1 \perp\!\!\!\perp R_3 \quad \Gamma'' \vdash e_3 : \mathbf{flip}^{R_3} \ ; \Gamma''' \quad R = R_2 \cup R_3}{\Gamma \vdash \mathbf{mux}(e_1, e_2, e_3) : \mathbf{flip}^R \times \mathbf{flip}^R \ ; \Gamma'''}$$

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

$$\textsc{Xor-Flip}\ \frac{\Gamma \vdash e_1 : \mathbf{bit}^{R_1}_{\ell_1} \ ; \Gamma' \quad \Gamma' \vdash e_2 : \mathbf{flip}^{R_2} \ ; \Gamma'' \quad R_1 \perp\!\!\!\perp R_2}{\Gamma \vdash \mathbf{xor}(e_1, e_2) : \mathbf{flip}^{R_2} \ ; \Gamma''} \qquad \textsc{Tup}\ \frac{\Gamma \vdash e_1 : \tau_1 \ ; \Gamma' \quad \Gamma' \vdash e_2 : \tau_2 \ ; \Gamma''}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \ ; \Gamma''}$$

$$\textsc{If}\ \frac{\Gamma' \vdash e_1 : \tau \ ; \Gamma_1'' \quad \Gamma \vdash e : \mathbf{bit}^{\varnothing}_\mathrm{P} \ ; \Gamma' \quad \Gamma' \vdash e_2 : \tau \ ; \Gamma_2''}{\Gamma \vdash \mathbf{if}(e)\{e_1\}\{e_2\} : \tau \ ; \Gamma_1'' \sqcup \Gamma_2''}$$

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

Fig. 5. $\lambda_{\mathbf{obliv}}$ Type System

$\Pr\left[\widetilde{x} \stackrel{?}{=} x\right]$ to indicate the probability that distribution $\widetilde{x}$ takes value x, and write $\lfloor x \rfloor$ as the point-distribution which gives probability 1 to x and 0 to all other elements of X. The function $\mathcal{S}(t)$ produces a distribution of traces that result from a single step of execution starting at the last element of t. When t has form $t \cdot e$ and e is a value, $\mathcal{S}$ copies the trace unmodified. The function $\mathcal{S}^*(\widetilde{t})$ lifts $\mathcal{S}$ to distributions of traces t. ($\mathcal{S}^*$ is the monadic extension of $\mathcal{S}$ in the probability monad $\mathcal{D}(\cdot)$, and embodies the law of total probability.) We write n repeated applications of $\mathcal{S}^*$ as $\mathcal{S}^*_n(\widetilde{t})$. As such, the probability of a particular length-n trace t where t is derived from the execution $e \rightsquigarrow_{q1} e_1 \rightsquigarrow_{q2} \ldots \rightsquigarrow_{qn} e_n$ would be $\Pr\left[\mathcal{S}^*_n(\lfloor e \rfloor) \stackrel{?}{=} t\right]$ and equal to $q1 \cdot q2 \ldots \cdot qn$.

3.3 Type System

Figure 5 defines the type system for $\lambda_{\mathbf{obliv}}$ 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'. Type environments map variables to either types τ or inaccessibility tags $\bullet$, explained shortly.

At a high level, the type system is enforcing three properties:

(1) No type errors occur at runtime (e.g., operating on a function as if it were a bit).

(2) No secrets can be inferred via direct or indirect flows
(3) No secrets can be inferred by probabilistic correlations between secrets and publicly revealed random numbers.

Properties (1) and (2) are enforced using standard techniques. For example, rule Mux-Bit only permits multiplexing on bits (e.g., not functions), and ensures that the security labels of the returned tuple are the join of the arguments to **mux**. (Recall that $S \sqcup \ell = S$ for all ℓ.) Property (3) is the main novelty of this work, and its need arises from $\lambda_{\textbf{obliv}}$'s support for random numbers that are used to enforce secrecy but can be later seen by the adversary.

A key invariant is that coin flips, which have type $\textbf{flip}^R$, always have the following properties:

A Their distribution is independent of all other values at flip type.
B Their distribution is *uniform*, meaning the probability of each possible bit value (**O** or **I**) is $1/2$.

The type system only allows creating, manipulating, and eliminating values at $\textbf{flip}^R$ type in ways that preserve (A) and (B). In particular, (A) is maintained by treating flip values as *affine*, which prevents their duplication, and (B) is maintained by tracking *probability regions* of flip values as they are combined and used in computations.

Affinity. To enforce non-duplicability, when an affine variable is used by the program, its type is removed from the residual environment. Figure 5 defines kinding metafunction $\mathcal{K}$ that assigns a type either the kind **U** (freely duplicatable) or **A** (non-duplicatable). Bits and functions are always universal, and flips are always affine; a pair is considered affine if either of its components is. Rule VarU in Figure 5 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.

There are a few other rules that are affected by affinity. Rules Cast-S and Cast-P permit converting $\textbf{flip}^R$ types to bits via the $\textbf{cast}_S$ and $\textbf{cast}_P$ coercions, respectively. The first converts a $\textbf{flip}^R$ to a $\textbf{bit}_S^R$ and does *not* make its argument inaccessible, while the second converts to a $\textbf{bit}_P^R$ and does make it inaccessible. In essence, the type system is enforcing that any random number is made adversary-visible at most once; secret copies are allowed because they are never revealed. The Fun rule ensures that no affine variables in the defining context are consumed within the body of the function, i.e., they are not captured by its closure. We write $\Gamma \uplus [x \mapsto \_, y \mapsto \_]$ to split a context into the part that binds x and y and Γ binds the rest; it is the Γ part that is returned, dropping the x and y bindings. Both Let and Let-Tup similarly remove their bound variables.

Finally, note that different variables could be made inaccessible in different branches of a conditional, so If types each branch in the same initial context, but then joins their the output contexts (as defined in Figure 5); if a variable is made inaccessible by one branch, it will be inaccessible in the joined environment.

Probability regions. Probability regions aim to enforce uniformity. A *probability region* ρ abstracts a set of coin flips produced at run-time, and *region set* R is a set of such regions. Region sets help track dynamic probabilistic dependencies between values, and the type rules prevent creating dependencies that would threaten uniformity. In particular, two coins are probabilistically independent if the intersection of their respective region sets is empty, which we write $R_1 \perp\!\!\!\perp R_2$. In other words, $R_1 \perp\!\!\!\perp R_2$ *iff* $R_1 \cap R_2 = \varnothing$

Rule Flip types a coin flip. The region set $\{\rho\} \cup R$ indicates that the generated coin should be associated with region ρ, and that we may expand with any other region set R as a form of weakening.[4] At creation, a coin's distribution is assumed uniform (i.e., bit values **O** and **I** are equally likely).

[4]The form of the annotation assures a coin's region set cannot be $\varnothing$.

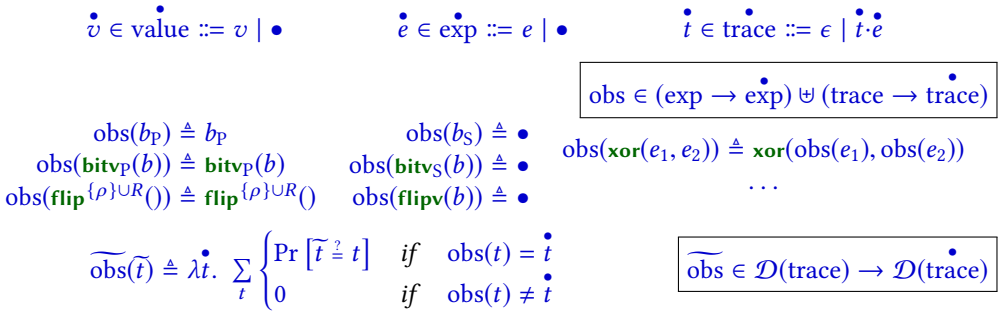


Fig. 6. Adversary observability

Per rule Bit, bit literals have a security label ℓ but probability region set $\emptyset$ as they are probabilistically independent of all other random values. Rule Cast-S crucially preserves the probability region set R from the input flip^R on the output type bit_S^R. As such, the type system can track the potential dependence of the converted value on other values. This information is used in rule Mux-Flip, which returns a pair of coins according the guard bit e_1. To ensure that the result of the mux remains uniform we require each coin's region sets to be independent of the guard's, i.e., $R_1 \perp\!\!\!\perp R_2$ and $R_1 \perp\!\!\!\perp R_3$. As such, the uniformity of the result is unaffected by possible correlation with R_1, so the output coins' region set R is $R_2 \cup R_3$ and not $R_1 \cup R_2 \cup R_3$.

These rules ensure the problematic example from the end of Section 2.4 (labeled (a), below) is rejected, as it could produce a non-uniform coin sk:

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

(a) Incorrect example

```
1  let sx = flip^ρ0 in
2  let sy,sz = mux(castS(sx),flip^ρ,flip^ρ)
```

(b) Correct example

The type checker first ascribes types flip^{ρ_1} and flip^{ρ_2} to sx and sy, respectively, according to rules Let-Tup, Flip, and Tup. It uses Cast-S to give $\text{castS}(\text{sx})$ type $\text{bit}_S^{\rho_1}$ and leaves sx accessible so that VarA can be used to give it and sy types flip^{ρ_1} and flip^{ρ_2}, respectively (then making them inaccessible). Rule Mux-Flip will now fail because the independence conditions do not hold. In particular, the region set ρ_1 of the guard is not independent of the region set ρ_1 of the second argument, i.e., $\rho_1 \perp\!\!\!\perp \rho_1$ does not hold. On the other hand, the program labeled (b) above is well-typed. Here, the Mux-Flip rule is happy because $\rho_0 \perp\!\!\!\perp \rho$. It is easy to see that regardless of sx's value, both sy and sz are uniformly distributed and independent of sx.

Rule Xor-Flip permits xor'ing a secret with a coin, returning a coin, as long as the secret's region set and the coin's region set are independent, which preserves uniformity. Rule Cast-P assigns region set $\emptyset$ to the output type, since it can no longer be used to influence the choice of a coin that was not already visible to the adversary. Rule Mux-Bit requires no independence checks—the output bits inherit the region set(s) of the arguments, and the join of their security labels. Finally, note that rule If requires the guard bit value to have label P since the execution trace reveals which branch is taken; branching on secrets must be done via mux.

3.4 Probabilistic Memory Trace Obliviousness

Figure 6 presents a model of the adversary's view of a computation as a new class of values, expressions and traces that "hide" sub-expressions (written $\bullet$) considered to be secret. We define obs for mapping ordinary expressions to adversary visible expressions, and likewise for values and traces. Secret bit expressions, secret bit values, and secret flip values all map to $\bullet$. Compound values and expressions call obs in recursive positions (*e.g.*, the xor case shown), and $\text{obs}(t\cdot e) \triangleq \text{obs}(t)\cdot\text{obs}(e)$.

We abbreviate obs in the figure and only show a small selection of cases. The distributional lifting of obs is defined as $\widetilde{\text{obs}}$ via the standard lifting from functions over sets to distributions.

Probabilistic memory trace obliviousness (PMTO) is then stated as the preservation of distributional equality modulo observation under any number of execution steps. We prove in the next section that well-typed $\lambda_{\textbf{obliv}}$ programs enjoy PMTO, stated formally as follows:

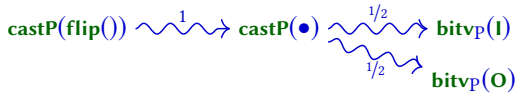
Proposition 3.1 (Probabilistic Memory Trace Obliviousness (PMTO)). *If* $\vdash e_1 : \tau, \vdash e_2 : \tau$ *and* $\text{obs}(e_1) = \text{obs}(e_2)$ *then for all* n, $\widetilde{\text{obs}}(\mathcal{S}^*_n(\lfloor e_1 \rfloor)) = \widetilde{\text{obs}}(\mathcal{S}^*_n(\lfloor e_2 \rfloor))$

4 PROVING PMTO

This section walks through our proof of PMTO. More details are in the Appendix.

Our overall approach is guided by the observation that a direct proof of PMTO on the sampling semantics ($\cdot \rightsquigarrow_q \cdot$, $\mathcal{S}$ or $\mathcal{S}^*$) is not straightforward because distinct traces and the adversary's view of them do not line up. Take for example the simple program that flips a coin and reveals that coin: $\textbf{cast}_{\textbf{P}}(\textbf{flip}())$. The probability-annotated transition DAG for this program is as follows:

The issue is in proving security w.r.t the adversary's view of the term, which is as follows:



In the adversary's view, the first step of the trace gets collapsed to one state because $\text{obs}(\textbf{flipv}(\textbf{I})) = \text{obs}(\textbf{flipv}(\textbf{O})) = \bullet$. After the revelation, the values become public, so the adversary observes a transition to two distinct public values, each with probability $1/2$. Because of this discrepancy between concrete semantics and adversary observation, it is not the case that $\mathcal{S}$ maps low-equivalent terms to low-equivalent distributions of terms, that is, $\text{obs}(\textbf{cast}_{\textbf{P}}(\textbf{flipv}(\textbf{I}))) = \text{obs}(\textbf{cast}_{\textbf{P}}(\textbf{flipv}(\textbf{O})))$, but $\widetilde{\text{obs}}(\mathcal{S}(\epsilon{\cdot}\textbf{cast}_{\textbf{P}}(\textbf{flipv}(\textbf{I})))) \neq \widetilde{\text{obs}}(\mathcal{S}(\epsilon{\cdot}\textbf{cast}_{\textbf{P}}(\textbf{flipv}(\textbf{O}))))$. However, the distributional version of the semantics $\mathcal{S}^*$ *does* preserve the security relation in this example, an in general for well-typed initial programs, in that the final distribution of terms are equal.

We resolve this discrepancy between the sampling semantics and the adversary's view of it via an alternative semantics that we call the *mixed* semantics. Under the assumptions of well-typedness, the mixed semantics both simulates the distributional sampling semantics $\mathcal{S}^*$ and faithfully models the adversary's view of computation. We call the semantics mixed because it embeds distributional structure into the syntax of values.

At a distance, our security proof proceeds by proving PMTO on the mixed semantics (§ 4.1 and 4.2), using its type system to establish key semantic invariants (§ 4.3 and 4.4), and then proving that the mixed semantics faithfully simulates the sampling semantics (§ 4.5) and the adversary's view of its events (§ 4.6), thus preserving the desired security property.

4.1 Intensional Distributions

Mixed semantics terms represent distributions of normal terms. We base a mixed term's representation on what we call *intensional distributions*, which overcome a limitation with the usual notion of distribution as a map from elements $x \in X$ to rational numbers between 0 and 1, which we will now refer to as the *extensional distribution* model.[5] Extensional distributions do not support probabilistic conditioning as an operation on distributional values, rather conditional probabilities must be

[5]We write x and y to range over elements of arbitrary sets X and Y; these are not program variables.

$$\overline{x} \in \mathcal{V}(Y) \triangleq \mathbb{N} \rightharpoonup Y \qquad \widehat{x} \in \mathcal{I}(X) \triangleq \mathcal{V}(\mathbb{B}) \rightharpoonup X \qquad \widehat{x} \otimes \widehat{y} \triangleq \lambda \overline{b}.\ \langle \widehat{x}(\overline{b}), \widehat{y}(\overline{b}) \rangle$$

$$\Pr\left[\widehat{x} \stackrel{?}{=} x\right] \triangleq \frac{\left|\left\{\overline{b} \mid \widehat{x}(\overline{b}) = x\right\}\right|}{\left|\left\{\overline{b} \mid \widehat{x}(\overline{b}) \text{ defined}\right\}\right|} \qquad \Pr\left[\widehat{x} \stackrel{?}{=} x \mid \widehat{y} \stackrel{?}{=} y\right] \triangleq \frac{\Pr\left[\widehat{x} \otimes \widehat{y} \stackrel{?}{=} \langle x, y \rangle\right]}{\Pr\left[\widehat{y} \stackrel{?}{=} y\right]}$$

$$\widehat{x} \perp\!\!\!\perp \widehat{y} \triangleq \forall x, y.\ \Pr\left[\widehat{x} \stackrel{?}{=} x \mid \widehat{y} \stackrel{?}{=} y\right] = \Pr\left[\widehat{x} \stackrel{?}{=} x\right]$$

$$\left[\widehat{x} \perp\!\!\!\perp \widehat{y} \mid \widehat{z} \stackrel{?}{=} z\right] \triangleq \forall x, y.\ \Pr\left[\widehat{x} \stackrel{?}{=} x \mid \widehat{y} \otimes \widehat{z} \stackrel{?}{=} \langle y, z \rangle\right] = \Pr\left[\widehat{x} \stackrel{?}{=} x \mid \widehat{z} \stackrel{?}{=} z\right]$$

Fig. 7. Intensional Distributions

stated axiomatically. That is, for any two distributions $\widetilde{x} \in \mathcal{D}(X)$ and $\widetilde{y} \in \mathcal{D}(Y)$, it is assumed by construction that the distributions are independent, *i.e.*, that $\Pr\left[\widetilde{x} \stackrel{?}{=} x \mid \widetilde{y} \stackrel{?}{=} y\right] = \Pr\left[\widetilde{x} \stackrel{?}{=} x\right]$.

The ability to model conditional probabilities between program variables is essential to our implementation of efficient oblivious data structures—the primary motivation for the design of $\lambda_{\textbf{obliv}}$. Consider the program **let** x = **flip()** **in** $\langle x, x \rangle$. After one evaluation step in the sampling semantics, the program will be reduced to a value $\langle b, b \rangle$ for some outcome of b of the coin flip. Our goal is to model this program as resulting in a pair of distributions (as opposed to a distribution of pairs) in order to support a substitution-based evaluation semantics. Using extensional distributions, it would be incorrect to model the result of the program as $\langle \text{coin}, \text{coin} \rangle$ since this model assumes each coin is independent, which is not the case. We develop intensional distributions which are cabable of encoding the result as effectively $\langle \text{coin}_1, \text{coin}_1 \rangle$, *i.e.*, two distributional values that "know" they are both the first coin, and that they're equal to each other. As such, our model supports encoding the result as a pair of distributions such that $\Pr\left[\text{coin}_1 \stackrel{?}{=} b\right] = 1/2$ yet the probability of the left coin conditioned on the right is uniquely determined, that is, $\Pr\left[\text{coin}_1 \stackrel{?}{=} b \mid \text{coin}_1 \stackrel{?}{=} b\right] = 1$.

As shown in Figure 7, an intensional distribution $\mathcal{I}(X)$ over a set X is a partial map from a vector of bits into X. Distributional elements are written $\widehat{x}$, and vectors $\mathcal{V}(Y)$ are partial maps from integers into Y. The vector in the domain of intensional distributions represents the set of coin flips that have taken place, i.e., the world of "entropy" in which the distribution is defined. For example, in a world in which two coins have been flipped, the first and second coins are encoded as $\widehat{c}_1 \triangleq \lambda \overline{b}.\ \overline{b}(1)$ and $\widehat{c}_2 \triangleq \lambda \overline{b}.\ \overline{b}(2)$ respectively. Unlike the extensional model, joint intensional distributions $\mathcal{I}(X \times Y)$ can be split into two distributions $\mathcal{I}(X)$ and $\mathcal{I}(Y)$ which may be still be correlated via the input entropy vector. The intensional distribution of pairs of a coin flip's outcome in $\mathcal{I}(\mathbb{B} \times \mathbb{B})$ would be $\lambda \overline{b}.\ \langle \overline{b}(1), \overline{b}(1) \rangle$, but could be split into a pair of distributions $\lambda \overline{b}.\ \overline{b}(1)$ and $\lambda \overline{b}.\ \overline{b}(1)$ and still maintain their connection to the first coin, and therefore each other.

Intensional distributions support a probability mass function and derived notions of conditional probability, probabilistic independence and conditional independence, as shown in the figure. The probability mass function must be treated with care. In Figure 7 we see that it is defined as the ratio of outcomes ($\{\overline{b} \mid \widehat{x}(\overline{b}) = x\}$) to the size of the domain of $\widehat{x}$ ($\{\overline{b} \mid \widehat{x}(\overline{b}) \text{ defined}\}$). This quantity is not well-defined for arbitrary partial maps $\mathcal{V}(\mathbb{B}) \rightharpoonup X$. The general requirement is a *continuity* condition, stating that the ratio stabilizes under the limit of N-length vectors as N goes to ∞. In our mechanized proof we track an explicit lower-bound N for each intensional distribution for which the probability function is fixed and therefore trivially converges for $N' \geq N$. In our on-paper description of the formalism, we only manipulate distribution maps for which this continuity condition holds as well. (We refer the reader to [Huang 2017] which works out the foundations of this intensional encoding in full generality.)

A key semantic fact about distributions used in our design of $\lambda_{\textbf{obliv}}$ is that the semantic conditional operation $\widehat{\text{cond}}(\widehat{b}, \widehat{y}, \widehat{z})$, is particularly well-behaved when (1) $\widehat{b}$ is independent of both $\widehat{y}$ and $\widehat{z}$, and

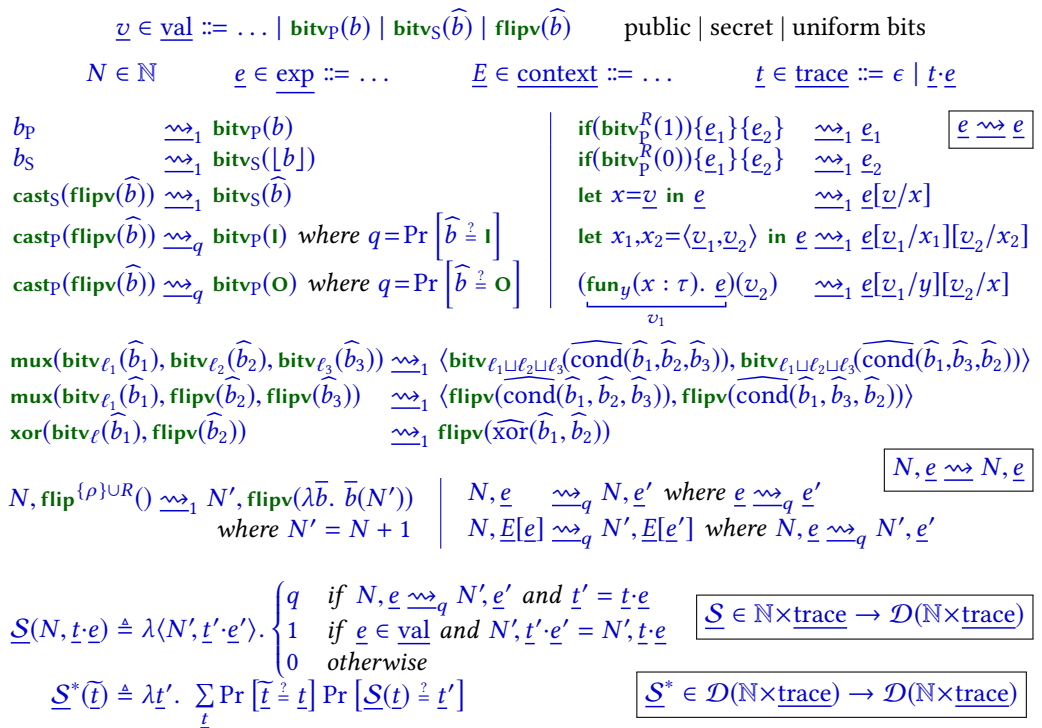


Fig. 8. Mixed Language Semantics

(2) $\widehat{y}$ and $\widehat{z}$ are equivalent distributions, written $\widehat{y} \approx \widehat{z}$, which means their probability mass functions coincide, so $\widehat{a} \approx \widehat{b} \triangleq \forall c.\ \Pr\left[\widehat{a} \stackrel{?}{=} c\right] = \Pr\left[\widehat{b} \stackrel{?}{=} c\right]$.

Lemma 4.1 (Bit Conditional Stability). *If* $\widehat{b} \perp\!\!\!\perp \widehat{x}$, $\widehat{b} \perp\!\!\!\perp \widehat{y}$ *and* $\widehat{x} \approx \widehat{y}$ *then* $\widehat{b} \perp\!\!\!\perp \widehat{\text{cond}}(\widehat{b}, \widehat{x}, \widehat{y})$ *and* $\widehat{\text{cond}}(\widehat{b}, \widehat{x}, \widehat{y}) \approx \widehat{x} \approx \widehat{y}$.

Proof. A simple calculation.[6] □

4.2 Mixed semantics

The mixed semantics is shown in Figure 8. The central idea is to use intensional distributions for secret bit/flip values, and non-distributional bits for public values. Mixed semantics expressions (and values, etc.) are underlined to distinguish them from the sampling semantics. The steps relation $N, e \rightsquigarrow_q N, e$ carries an extra piece of state—N—which encodes the identity of the next coin flip. All configuration rules except two do not manipulate N. The flip rule creates a value which is the N-th coin flip (i.e., the N-th element of the intensional distribution's input vector $\overline{b}$) after incrementing N, and the contextual reduction rule on $E[e]$ propagates changes to N.

In the mixed semantics, flip expressions always produces a single uniform distributional value, and the step is taken with probability annotation $q = 1$. A secret literal will produce a point distribution on that literal. Point distributions are written $\lfloor x \rfloor$ for $\lambda\overline{b}.\ x$. Semantic operations for **mux** and **xor** are lifted to manipulate distributions of secrets, e.g., $\widehat{\text{xor}}(\widehat{c}_1, \widehat{c}_2) \triangleq \lambda\overline{b}.\ \text{xor}(\widehat{c}_1(\overline{b}), \widehat{c}_2(\overline{b}))$.

[6] $\Pr\left[\widehat{\text{cond}}(\widehat{b}, \widehat{x}, \widehat{y}) \stackrel{?}{=} b\right] = \Pr\left[\widehat{b} = \mathbf{I}\right] \Pr\left[\widehat{x} \stackrel{?}{=} b \mid \widehat{b} = \mathbf{I}\right] + \Pr\left[\widehat{b} = \mathbf{O}\right] \Pr\left[\widehat{y} \stackrel{?}{=} b \mid \widehat{b} = \mathbf{O}\right] = \Pr\left[\widehat{b} = \mathbf{I}\right] \Pr\left[\widehat{x} \stackrel{?}{=} b\right] + \Pr\left[\widehat{b} = \mathbf{O}\right] \Pr\left[\widehat{y} \stackrel{?}{=} b\right]$
$= \Pr\left[\widehat{b} = \mathbf{I}\right] \Pr\left[\widehat{x} \stackrel{?}{=} b\right] + \Pr\left[\widehat{b} = \mathbf{O}\right] \Pr\left[\widehat{x} \stackrel{?}{=} b\right] = (\Pr\left[\widehat{b} = \mathbf{I}\right] + \Pr\left[\widehat{b} = \mathbf{O}\right]) \Pr\left[\widehat{x} \stackrel{?}{=} b\right] = \Pr\left[\widehat{x} \stackrel{?}{=} b\right] = \Pr\left[\widehat{y} \stackrel{?}{=} b\right]$

$\Psi \in \text{rtyping} \triangleq \mathbb{N} \rightharpoonup \text{rvar}$ (region typing) $\quad \Phi \in \text{pub-kn} \triangleq \wp(\mathcal{I}(\mathbb{B}))$ (public knowledge)

$\boxed{\Psi, \Phi \vdash \underline{v} : \tau}$

$$\textsc{BitS}\ \frac{\forall n \in \mathrm{dom}(\Psi).\ \Psi(n) \in R \vee \left[\widehat{b} \perp\!\!\!\perp (\lambda \overline{b}.\, \overline{b}(n)) \;\middle|\; \Phi\right]}{\Psi, \Phi \vdash \mathbf{bit_S}(\widehat{b}) : \mathbf{bit_S}^{R}} \qquad \textsc{Flip}\ \frac{\forall n \in \mathrm{dom}(\Psi).\ \Psi(n) \in R \vee \left[\widehat{b} \perp\!\!\!\perp (\lambda \overline{b}.\, \overline{b}(n)) \;\middle|\; \Phi\right] \quad \forall b.\ \Pr\left[\widehat{b} \stackrel{?}{=} b \;\middle|\; \Phi\right] = 1/2}{\Psi, \Phi \vdash \mathbf{flip}^{R}(\widehat{b}) : \mathbf{flip}^{R}} \qquad \textsc{BitP}\ \frac{}{\Psi, \Phi \vdash \mathbf{bit_P}(b) : \mathbf{bit_P}^{\varnothing}}$$

Fig. 9. Mixed Semantics Typing

Other operations are as usual, e.g., let expressions and tuple elimination reduce via substitution and are not lifted to distributions. Because public bits are not distributional, the mixed semantics has two rules for $\mathbf{cast_P}$: one reduces its distributional argument $\mathbf{flipv}(\widehat{b})$ to $\mathbf{bitv_P}(\mathbf{I})$, annotating the arrow with the probability q that $\widehat{b}$ is $\mathbf{I}$, and the other rule likewise reduces $\mathbf{flipv}(\widehat{b})$ to $\mathbf{bitv_P}(\mathbf{O})$.

The metafunction $\widehat{\text{cond}}$ is the bitwise conditional lifted to distributions. The rules for **mux** and **xor** are slightly imprecise in that they are not specialized to all 8 possible combinations of public and secret arguments. We instead write the pattern $\mathbf{bitv}_\ell(\widehat{b})$ as shorthand for $\widehat{b} = \widehat{b'}$ when the argument is $\mathbf{bitv_S}(\widehat{b'})$ and $\widehat{b} = \lfloor b' \rfloor$ when the argument is $\mathbf{bitv_P}(b')$. In the event that all arguments are public, the result will be a point distribution and is lowered to a non-distributional value, *e.g.*, the rule $\mathbf{mux}(\mathbf{bitv_P}(b_1), \mathbf{bitv_P}(b_2), \mathbf{bitv_P}(b_2)) \rightsquigarrow_1 \mathbf{bitv_P}(\mathrm{cond}(b_1, b_2, b_3))$ is implied, where cond is the unlifted bitwise conditional.

Notice the symmetry in the mixed semantics compared to the sampling semantics: there are two flip rules in the sampling semantics, each with 1/2 probability, while there are two $\mathbf{cast_P}$ rules in the mixed semantics, each with 1/2 probability (assuming that flip values are uniformly distributed, which we will prove is always the case). This structure more accurately mirrors the adversary's view of a trace in the sampling semantics.

4.3 Mixed Semantics Typing

Figure 9 defines the typing rules for mixed semantics terms. Two new structures are introduced for value typing: a region type environment Ψ and a set of public knowledge Φ. The region type environment captures the static region in which every coin flip is generated in the dynamic semantics. The set of coin flips that live in a region set R is $\{n \mid \Psi(n) \in R\}$. Consider the program $\mathbf{mux}(\mathbf{flip}^{\rho_1}(), \mathbf{flip}^{\rho_2}(), \mathbf{flip}^{\rho_2}())$, which flips one coin in ρ_1 and two in ρ_2. The resulting region type environment is $\Psi = \{1 \Rightarrow \rho_1, 2 \Rightarrow \rho_2, 3 \Rightarrow \rho_2\}$. The set of public knowledge contains bit distributions $\widehat{b}$ under which semantic independence requirements are conditioned, as explained shortly.

Key invariants are stated directly by the mixed semantics type system, namely that (1) coins are always uniformly distributed, and (2) that random values in non-overlapping region sets are probabilistically independent. Invariant (2) is shown in the premise of BitS and the first premise of Flip and (1) is shown in the second premise of Flip. The non-overlapping region sets premise states that for each coin flipped thus far, that coin is either contained in the static region of the value—meaning there may be a probabilistic dependence on that coin—or that coin is probabilistically independent of the value. The independence property is conditioned by public knowledge Φ because it may not be that the values are universally independent, but only independent after conditioning on prior revelations. Consider a program that reveals a flipped value, which then influences a secret random value.

This example is typeable in our system and demonstrates how secret random values are assumed to not influence the distributions of other secret random values after they are revealed to public with $\mathbf{cast}_\mathrm{P}$. To typecheck each step of the execution of this program, the fact that the coin was revealed is added to the set of public knowledge after the second line, *e.g.*, $\Phi = \{\widehat{b}_c \stackrel{?}{=} \mathbf{1}\}$ where $\mathbf{bitv}_\mathrm{S}(\widehat{b}_c)$ is the value substituted for program variable c and the $\mathbf{1}$ is the revealed bit. We must track this public knowledge in order for the value of y to be well-typed in region ρ_2, which is only true if y is probabilistically independent of coins which live in all other regions, including ρ_1. This is not universally true: y is technically dependent on region ρ_1 because it depends on c, however y is independent of c after conditioning on the outcome of the revelation at line 2. For this reason we must allow for conditioning on Φ in the typing rules for mixed values. This conditioning is acceptable because Φ only reflects revelations made publicly visible to the adversary.

$$\begin{array}{l}\mathbf{let}\ c, s = \ldots \quad : \mathbf{flip}^{\rho_1} \times \mathbf{bit}_\mathrm{S}^{\rho_2} \\ \mathbf{let}\ x = \mathbf{cast}_\mathrm{P}(c) \ : \mathbf{bit}_\mathrm{P}^{\varnothing} \\ \mathbf{let}\ y = \mathbf{xor}(x, s) : \mathbf{bit}_\mathrm{S}^{\rho_2}\end{array}$$

Typing for mixed values ensures a semantic property for regions: secret values which are typeable in non-overlapping static region sets are probabilistically independent.

Lemma 4.2 (Static Region Soundness). *If* $\Psi, \Phi \vdash \mathbf{bit}_\mathrm{S}(\widehat{b}_1) : \mathbf{bit}_\mathrm{S}^{R_1}$, $\Psi, \Phi \vdash \mathbf{bit}_\mathrm{S}(\widehat{b}_2) : \mathbf{bit}_\mathrm{S}^{R_2}$ *and* $R_1 \cap R_2 = \varnothing$, *then* $\left[\widehat{b}_1 \perp\!\!\!\perp \widehat{b}_2 \;\middle|\; \Phi\right]$.

Proof. Using the hypothesis of the typing rule for flip values we construct sets of coins θ_1 and θ_2 such that $\widehat{b}_1$ depends uniquely on θ_1 and $\widehat{b}_2$ depends uniquely on θ_2 (conditioned on public knowledge Φ). By $R_1 \cap R_2 = \varnothing$ and functionality of Ψ we know that θ_1 and θ_2 are non-overlapping, and thus $\widehat{b}_1$ and $\widehat{b}_2$ are independent conditioned on public knowledge Φ. □

We prove similar results for relating secret bits to flip values, and flip values to each other. (The property is not used for public bits.)

We design typing for flip values to ensure that type safety (progress & preservation) for the mixed semantics will imply the two key semantics invariants (region independence and uniformity) needed to establish PMTO. To extend typing to expressions with embedded runtime values, we reinterpret the original typing rules $\Gamma \vdash e : \tau \;;\; \Gamma'$ as $\Psi, \Phi, \Gamma \vdash e : \tau \;;\; \Gamma'$ for fixed Ψ and Φ.

4.4 Type Safety via Well-partitioning

We are not able to prove type safety for the mixed semantics with what we have seen so far. In particular, establishing that non-overlapping region sets implies probabilistic independence on its own is insufficient. The issue is ensuring that public revelations do not perturb the uniform distributions of other flip values (invariant (2) above). This property is not maintained via regions, and has the potential to be violated during revelations. Consider the following (ill-typed) program which copies a flip value which is later revealed.

$$\begin{array}{l}\mathbf{let}\ x = \mathbf{flip}() \\ \mathbf{let}\ y = x \\ \ldots\ \mathbf{cast}_\mathrm{P}(x)\ \ldots\ y\ \ldots\end{array}$$

In order for the term to be well-typed after line 3, the value substituted for variable y must be uniformly distributed *conditioned on public information* Φ (per the Flip rule in Figure 9). After the revelation, Φ will contain a condition for the outcome of x, which will in turn uniquely determine the value of y—thus y is no longer uniform; rather, it is a point distribution after conditioning on Φ.

We address this issue in the type system by treating public revelations affinely; in the example the $\mathbf{cast}_\mathrm{P}(x)$ on line 3 is rejected because x was made inaccessible on line 2 when assigned to y. To show that affinity is sufficient to ensure invariant (2), i.e., that public revelations do not problematically influence other flip values, we develop a semantic invariant called *well-partitioning* which means

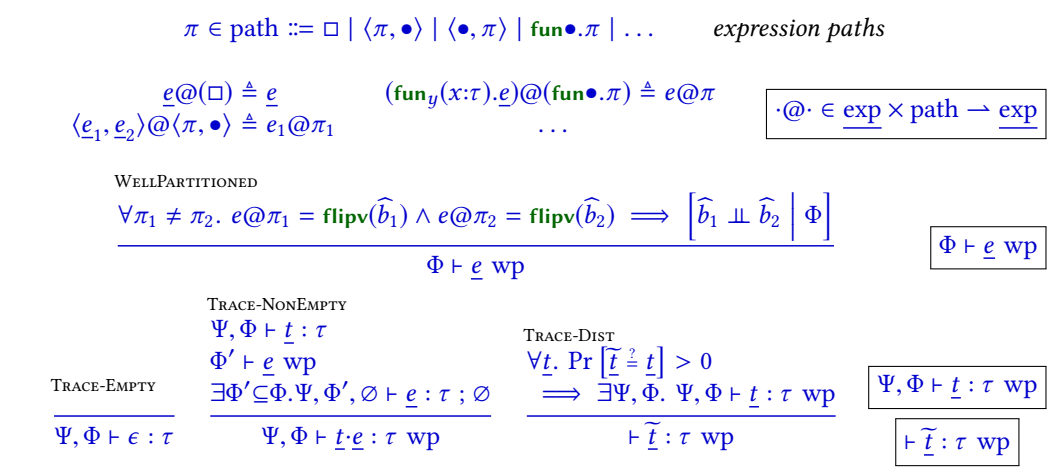


Fig. 10. The Well-partitioning Property Enforced by Affinity

that for every two flip values which occur at different sub-expression positions, those two flip values are independent conditioned on Φ. We formalize this notion via a syntactic category for *paths* into expressions (similar to a reduction context, but which descends into all sub-expressions), as shown in Figure 17. Also in the figure is a combined well-typedness and well-partitionedness judgment for traces and trace distributions due to their shared dependence on public knowledge Φ, which is existentially quantified in the distributional judgment.

We then prove type safety for the distributional trace semantics under the additional assumption of well-typing and well-partitioning.

Lemma 4.3 (Mixed Semantics Progress). *If* $\Psi, \Phi \vdash \underline{e} : \tau$ *then either* $\underline{e}$ *is a value or for any* $N > \lceil \text{dom}(\Psi) \rceil$[7] *then there exists* $N' \geq N, \underline{e}'$ *and* q *s.t.* $N, \underline{e} \underline{\rightsquigarrow}_q N', \underline{e}'$.

Proof. By induction on the typing derivation. □

Lemma 4.4 (Mixed Semantics Preservation). *If* $\Psi, \Phi, \varnothing \vdash \underline{e} : \tau \,;\, \varnothing$, $\Phi \vdash\!\downarrow -e$ wp *and* $N, \underline{e} \underline{\rightsquigarrow}_q N', \underline{e}'$ *then there exists* $\Psi' \supseteq \Psi$ *and* $\Phi' \supseteq \Phi$ *s.t.* $\Psi', \Phi' \vdash \underline{e}' : \tau$ *and* $\Phi' \vdash \underline{e}'$ wp.

Proof. By induction on the typing derivation and case analysis on the steps relation. MuxBit is justified by Lemma 4.2. MuxFlip is justified by Lemma 4.2, Lemma 4.1, and appeals to well-partitioning. CastP is justified by well-partitioning. □

Type safety for the distributional trace semantics is a corollary of progress and preservation.

Corollary 4.5 (Mixed Trace Semantics Type Safety). *If* $\vdash \widetilde{\underline{t}} : \tau$ wp *then* $\underline{\mathcal{S}}^*(\widetilde{\underline{t}})$ *is a proper distribution (sums to one) and* $\vdash \underline{\mathcal{S}}^*(\widetilde{\underline{t}}) : \tau$.

We also prove analogous type safety results for the sampling semantics. Because the sampling semantics does not encode any semantic invariants, the proofs are comparably simpler.

Lemma 4.6 (Sampling Semantics Type Safety). *If* $\varnothing \vdash e : \tau \,;\, \varnothing$ *then either* e *is a value or there exists* e' *and* q *s.t* $e \rightsquigarrow_q e'$ *and* $\varnothing \vdash e' : \tau \,;\, \varnothing$.

Corollary 4.7 (Sampling Trace Semantics Type Safety). *If* $\vdash \widetilde{t} : \tau$ *then* $\mathcal{S}^*(\widetilde{t})$ *is a proper distribution (sums to one) and* $\vdash \mathcal{S}^*(\widetilde{t}) : \tau$.

$$\widehat{\mathcal{U}}(x) \triangleq \lambda\overline{b}.\ x \qquad \widehat{\mathcal{U}}(\mathbf{bitv}_\ell(\widehat{b})) \triangleq \lambda\overline{b}.\ \mathbf{bitv}_\ell(\widehat{b}(\overline{b}))$$
$$\widehat{\mathcal{U}}(b_\ell) \triangleq \lambda\overline{b}.\ b_\ell \qquad \widehat{\mathcal{U}}(\mathbf{flipv}(\widehat{b})) \triangleq \lambda\overline{b}.\ \mathbf{flipv}(\widehat{b}(\overline{b}))$$

$$\widehat{\mathcal{U}}(\mathbf{xor}(\underline{e}_1, \underline{e}_2)) \triangleq \lambda\overline{b}.\ \mathbf{xor}(\widehat{\mathcal{U}}(\underline{e}_1)(\overline{b}), \widehat{\mathcal{U}}(\underline{e}_2)(\overline{b}))$$
$$\widehat{\mathcal{U}}(\mathbf{fun}_y(x:\tau).\ \underline{e}) \triangleq \lambda\overline{b}.\ \mathbf{fun}_y(x:\tau).\ \widehat{\mathcal{U}}(\underline{e})(\overline{b})$$

$$\ldots \qquad \boxed{\widehat{\mathcal{U}} \in \underline{\mathrm{exp}} \to \mathcal{I}(\mathrm{exp})}$$

$$\widehat{\mathcal{U}}(\epsilon) \triangleq \lambda\overline{b}.\ \epsilon \qquad \widehat{\mathcal{U}}(\underline{t}\cdot\underline{e}) \triangleq \lambda\overline{b}.\ \widehat{\mathcal{U}}(\underline{t})(\overline{b})\cdot\widehat{\mathcal{U}}(\underline{e})(\overline{b}) \qquad \boxed{\widehat{\mathcal{U}} \in \underline{\mathrm{trace}} \to \mathcal{I}(\mathrm{trace})}$$

$$\widetilde{\mathcal{U}}(\underline{t}) \triangleq \lambda t.\ \Pr\left[\widehat{\mathcal{U}}(\underline{t}) \stackrel{?}{=} t\right] \qquad \boxed{\widetilde{\mathcal{U}} \in \underline{\mathrm{trace}} \to \mathcal{D}(\mathrm{trace})}$$

$$\widetilde{\mathcal{U}}^*(\widetilde{t}) \triangleq \lambda t.\ \sum_{\underline{t}} \Pr\left[\widetilde{t} \stackrel{?}{=} \underline{t}\right] \Pr\left[\widetilde{\mathcal{U}}(\underline{t}) \stackrel{?}{=} t\right] \qquad \boxed{\widetilde{\mathcal{U}}^* \in \mathcal{D}(\underline{\mathrm{trace}}) \to \mathcal{D}(\mathrm{trace})}$$

Fig. 11. Mixed Semantics Simulation

4.5 Simulation

Now that we have established all key semantic invariants needed to prove PMTO, the next step is to relate the mixed semantics to the normal semantics via a simulation argument. To do this, Figure 18 defines a function $\widetilde{\mathcal{U}}$ which maps individual mixed semantics terms to an extensional distribution of sampling semantics terms. This lifting occurs via a lifting $\widehat{\mathcal{U}}$ to intensional distributions, from there via $\widetilde{\mathcal{U}}$ to extensional ones, and finally via $\widetilde{\mathcal{U}}^*$ to mixed traces (similarly to what was done for $\mathcal{S}^*$). In the presence of well-typedness, equality through $\mathcal{U}$ is maintained throughout evaluation.

Lemma 4.8 (Mixed Semantics Simulation). *If* $\vdash \widetilde{t} : \tau$, $\widetilde{t} = \widetilde{\mathcal{U}}(\underline{t})$ *and* $\vdash \underline{t} : \tau$ wp *then* $\mathcal{S}^*(\widetilde{t}) = \widetilde{\mathcal{U}}^*(\underline{\mathcal{S}}(\underline{t}))$.

Proof. By induction on typing for mixed terms and case analysis on the steps relation. □

A simple corollary is that the lifted distributional semantics $\mathcal{S}^*$ also preserves equality under $\widetilde{\mathcal{U}}^*$.

4.6 Low-equivalence Preservation

Now that we have shown that the mixed semantics simulates the sampling semantics, it suffices to prove a non-interference lemma for mixed semantics evaluation, and that non-interference of mixed semantics terms implies non-interference for distributions of sampling semantics terms.

We show the low-equivalence relation for mixed terms in Figure 19. The axioms of the relation are reflexivity—every term is related to itself–and axioms which allow secret expressions and values to be related to any other secret. The congruence rules allow for these axioms to appear in any subexpression position. Traces are related pointwise, and distributions of traces are related if for every trace in one distribution, there exists a related trace in the other with the same probability. Note that the relation $\widetilde{t} \sim \widetilde{t}$ is symmetric; the proof of this fact exploits the fact that distributions sum to 1. In the presence of well-typing and well-partitioning, the mixed semantics preserves the low-equivalence relation under evaluation.

Lemma 4.9 (Low-equivalence Preservation). *If* $\Psi, \Phi, \varnothing \vdash \underline{e}_1 : \tau \,;\, \varnothing$, $\Phi \vdash \underline{e}_1$ wp, $\Psi, \Phi, \varnothing \vdash \underline{e}_2 : \tau \,;\, \varnothing$ $\Phi \vdash \underline{e}_2$ wp *and* $\underline{e}_1 \sim \underline{e}_2$, *then* $\underline{\mathcal{S}}(\underline{e}_1) \sim \underline{\mathcal{S}}(\underline{e}_2)$.

Proof. By induction on the derivation of low-equivalence and appeals to typing derivations and well-partitioned derivations for $\underline{e}_1$ and $\underline{e}_2$. □

[7] $\lceil \mathrm{dom}(\Psi) \rceil$ is the largest number in the domain of Ψ

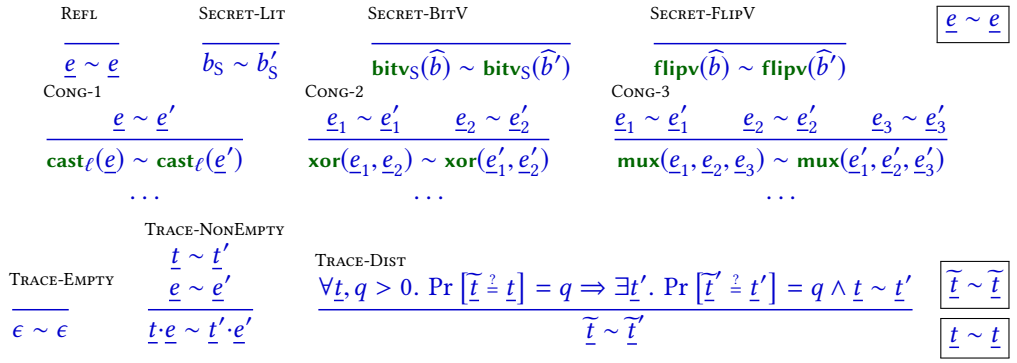


Fig. 12. Low-equivalence for Mixed Terms

A simple corollary is that the lifted semantics $\underline{\mathcal{S}}^*$ also preserves the low-equivalence relation.

To relate low-equivalence back to the sampling semantics, we prove soundness of the judgment w.r.t equality modulo adversary observation.

Lemma 4.10 (Low-equivalence Soundness). *If* $\Psi, \Phi, \varnothing \vdash \underline{e}_1 : \tau \,;\, \varnothing$, $\Psi, \Phi, \varnothing \vdash \underline{e}_2 : \tau \,;\, \varnothing$ *and* $\underline{e}_1 \sim \underline{e}_2$, *then* $\widetilde{\mathrm{obs}}(\mathcal{U}(\underline{e}_1)) = \widetilde{\mathrm{obs}}(\mathcal{U}(\underline{e}_2))$.

Proof. By induction on the derivation of low-equivalence. □

4.7 Proving PMTO

We can now present the final proof, briefly recapping the overall setup. First, we extended the notion of trace t of normal terms to traces $\underline{t}$ of mixed terms—terms with intensional distributional values embedded inside (§4.1). The mixed semantics is also probabilistic, inducing a distributional view of the mixed semantics $\underline{\mathcal{S}}$ and its lifting $\underline{\mathcal{S}}^*$ (§4.2). We establish key semantic invariants about runtime values in the mixed semantics, and enforce them via typing and a well-partitioning property (§4.3 and §4.4). Each mixed trace $\underline{t}$ represents a distribution of normal traces, which we capture through a mapping $\mathcal{U}$ (§4.5). We also develop a low-equivalence relation for mixed terms which captures the semantics of PMTO security (§4.6). The proof is essentially an argument that evaluation for distributions of terms preserves each of these invariants and relationships.

The final PMTO argument for a single pair of low-equivalent source programs is "kicked off" by injecting them into the mixed semantics, which is possible because the two language coincide in the absence of runtime values. These injected mixed terms will trivially satisfy the stronger invariants of the mixed semantics, as well as a relationship with their non-mixed selves through $\mathcal{U}$. Each term is allowed to evaluate for any number of steps, resulting in a distribution of traces. The resulting distribution of mixed traces will be low-equivalent, which implies that the resulting normal distributions are low equivalent as well, which establishes PMTO.

Before proving PMTO, we first prove a strong, one-step variant of PMTO which is used transitively in the final PMTO proof.

Lemma 4.11 (Strong One-step PMTO).

$$\textit{If}\quad \begin{array}{c} \widetilde{\underline{e}}_1 \sim \widetilde{\underline{e}}_2 \\ \varnothing \vdash \widetilde{\underline{e}}_1 : \tau \,;\, \varnothing \\ \varnothing \vdash \widetilde{\underline{e}}_2 : \tau \,;\, \varnothing \\ \Psi, \Phi, \varnothing \vdash \widetilde{\underline{e}}_1 : \tau \,;\, \varnothing \\ \Psi, \Phi, \varnothing \vdash \widetilde{\underline{e}}_2 : \tau \,;\, \varnothing \end{array} \quad \begin{array}{c} \widetilde{e}_1 = \widetilde{\mathcal{U}}^*(\widetilde{\underline{e}}_1) \\ \widetilde{e}_2 = \widetilde{\mathcal{U}}^*(\widetilde{\underline{e}}_2) \\ \Phi \vdash \widetilde{\underline{e}}_1 \text{ wp} \\ \Phi \vdash \widetilde{\underline{e}}_2 \text{ wp} \end{array} \quad \textit{then} \quad \begin{array}{c} \underline{\mathcal{S}}^*(\widetilde{\underline{e}}_1) \sim \underline{\mathcal{S}}^*(\widetilde{\underline{e}}_2) \\ \vdash \mathcal{S}^*(\widetilde{e}_1) : \tau \\ \vdash \mathcal{S}^*(\widetilde{e}_2) : \tau \end{array} \quad \begin{array}{c} \mathcal{S}^*(\widetilde{e}_1) = \widetilde{\mathcal{U}}^*(\underline{\mathcal{S}}^*(\widetilde{\underline{e}}_1)) \\ \mathcal{S}^*(\widetilde{e}_2) = \widetilde{\mathcal{U}}^*(\underline{\mathcal{S}}^*(\widetilde{\underline{e}}_2)) \\ \vdash \underline{\mathcal{S}}^*(\widetilde{\underline{e}}_1) : \tau \text{ wp} \\ \vdash \underline{\mathcal{S}}^*(\widetilde{\underline{e}}_2) : \tau \text{ wp} \end{array}$$

Proof. Follows from Lemmas 4.5, 4.7, 4.8 and 4.9. □

Theorem 4.12 (Probabilistic Memory Trace Obliviousness (PMTO)). *For source program terms* e_1 *and* e_2*, if* $\varnothing \vdash e_1 : \tau \; ; \varnothing$, $\varnothing \vdash e_2 : \tau \; ; \varnothing$ *and* $\mathsf{obs}(e_1) = \mathsf{obs}(e_2)$ *then for all* n, $\widetilde{\mathsf{obs}}(\mathcal{S}_n^*(\lfloor e_1 \rfloor)) = \widetilde{\mathsf{obs}}(\mathcal{S}_n^*(\lfloor e_2 \rfloor))$

Proof. By transitive applications of Lemma 4.11 initiated with $\widetilde{e}_1 = \widetilde{\underline{e}}_1 = \lfloor e_1 \rfloor$ and $\widetilde{e}_2 = \widetilde{\underline{e}}_2 = \lfloor e_2 \rfloor$ and concluding with application of Lemma 4.10. □

5 CASE STUDY: TREE-BASED ORAM

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

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

To support implementing Tree ORAM, our implementation extends $\lambda_{\mathbf{obliv}}$ in various ways. Most notably, we support arrays and records. Arrays have kind `U` (universal), meaning they are not treated affinely. Arrays have three operations. First, a `length(a)` operation which returns the length of the array as a `nat P`. Second, reading from an array `a[n]` which will evaluate to the `n`th entry of the array `a`. Third, a simultaneous read/write operation `a[n] ← e` which writes `e` to the `n`th entry of `a`, but returns what was in `a[n]` prior to this write. The index into the array for both the read and write operations must be `nat P`. When the contents of the array are `A` (affine), only the simultaneous read/write operation is well-typed, since reading out the contents of the array slot will consume them, and so they must be replaced with something fresh.

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

Our implementation supports *region polymorphism* wherein we may specify region variables and then substitute concrete regions for them so long as the substituted regions satisfy programmer-specified independence constraints. We do not support data polymorphism, although we believe it is a delicate but standard extension. In what follows, we write `nat S` to be a secret number in region $\emptyset$; `nat P` for a public number in $\emptyset$; `nat R` to be a secret in the polymorphic region `R`. We also write type `rnd R` to be a random natural number in the polymorphic region `R`; in our implementation we encode these using tuples of **`flip`** `R`s.

5.1 Non-recursive ORAM (NORAM)

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

```
type noram R R' = bucket array R R'
```

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

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

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

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

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

let trivial_rr (troram: bucket) (idx: nat R) : rnd R' * rnd R' =
  let ret: block = trivial_rr_h troram idx 0 dummy_block() in
  ret.data

let rec noram_rr_h (noram: noram) (idx: nat R) (tag: nat P) (level: nat P) (acc: block): block =
  (* compute the first index into the bucket array at depth level *)
  let base: nat P = (pow 2 level) - 1 in
  if base >= length(noram) then acc
  else
    let bucket_loc: nat P = base + (tag & base) in (* the bucket on the path to access *)
    let bucket = noram[bucket_loc] in
    let acc = trivial_rr_h bucket idx 0 acc in
    noram_rr_h noram idx tag (level + 1) acc

let noram_rr (noram: noram) (idx: nat R) (tag: nat P): rnd R' * rnd R' =
  let ret = noram_rr_h noram idx tag 0 dummy_block() in
  ret.data
```

The noram_rr function takes the noram as its first argument, and the index idx of the desired element as its second. The tag argument is the position tag, which identifies a path through the noram binary tree along which the indexed value will be stored, if present. This argument has type nat P because it (or derivatives of it) will be used index the arrays that make up the NORAM, and these indexes are adversary-visible. The noram_rr function works by calling noram_rr_h which recursively works its way down the identified path. It maintains an accumulator, acc: 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: rnd R' * rnd R'. This member of the record must be generated fresh for each new block, since its contents are treated affinely. Each recursive call to noram_rr_h

moves to a node the next level down in the tree, as determined by the tag. At each node, it reads out the bucket array, which as mentioned earlier is essentially a Trivial ORAM. The trivial_rr function calls trivial_rr_h to iterate through the entire bucket, to obliviously read out the desired block, if present.

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

```
noram_add: noram → (idx: nat R) → (tag: nat R) → (data: nat R' * nat R') → unit
```

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

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

5.2 Full Recursive ORAM

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

As mentioned above, one way to implement the position map is to just use a regular array stored in hidden memory, e.g., on-chip in a secure processor deployment of ORAM. However, this is not possible for MPC-based deployments, the adversary can observe the access pattern on the map itself. To avoid this problem, we can use another smaller Tree-based ORAM to store PM. In particular, the ORAM PM has N/c blocks, where each block contains c elements, where $c \geq 2$ is a parameter

of the ORAM. Therefore, PM also contains another position map PM ′, which can be stored as an ORAM of capacity N/c^2. Such a construction can continue recursively, until the position map at the bottom is small enough to be constructed as a Trivial ORAM. Therefore, there are $\log_c N$ norams constructed to store those position maps, and thus the overall runtime of a recursive ORAM is $\log_c N$ times of the runtime of a noram. In the following, we implement the ORAM with $c = 2$.[8]

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

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

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

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

```
1  let rec tree_rr_h (oram: oram) (idx: nat S) (level: nat P): rnd R' * rnd R' =
2    let (norams, troram) = oram in
3    let levels: nat P = length(norams) in
4    if level >= levels then trivial_rr troram idx
5    else
6      let (r0, r1): rnd R' * rnd R' = tree_rr_h oram (idx / 2) (level + 1) in
7      let (r0', tag) = mux(idx % 2 = 0, rnd, 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: nat S): rnd R' * rnd R' =
13   tree_rr_h oram idx 0
```

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

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

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

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

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

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

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

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

6 RELATED WORK

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

- Processor architectures that mitigate leakage through timing [Kocher et al. 2004; Liu et al. 2012], power consumption [Kocher et al. 2004], or memory-traces [Fletcher et al. 2014; Liu et al. 2015a; Maas et al. 2013; Ren et al. 2013].
- Program analysis techniques that formally ensure that a program has bounded or no leakage through instruction traces [Molnar et al. 2006], timing channels [Agat 2000; Molnar et al. 2006; Russo et al. 2006; Zhang et al. 2012, 2015], or memory traces [Liu et al. 2015a, 2013, 2014].
- Algorithmic techniques that transform programs and algorithms to their side-channel-mitigating or side-channel-free counterparts while introducing only mild costs – e.g., works on mitigating timing channel leakage [Askarov et al. 2010; Barthe et al. 2010; Zhang et al. 2011], and on preventing memory-trace leakage [Blanton et al. 2013; Eppstein et al. 2010; Goldreich 1987; Goldreich and Ostrovsky 1996; Goodrich et al. 2012; Shi et al. 2011; Stefanov et al. 2013; Wang et al. 2015, 2014; Zahur and Evans 2013].

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

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

ObliVM [Liu et al. 2015b] is a language for programming oblivious algorithms intended to be run as secure multiparty computations [Yao 1986]. Its type system also employs affine types to ensure random numbers are used at most once. However, there it provides no mechanism to

disallow constructing a non-uniformly distributed random number. When such random numbers are generated, they can be distinguished by an attacker from uniformly distributed random numbers when being revealed. Therefore, the type system in ObliVM does not guarantee obliviousness. $\lambda_{\mathbf{obliv}}$'s use of probability regions enforces that all random numbers are uniformly random, and thus eliminates this channel of information leakage. Moreover, we prove that this mechanism (and the others in $\lambda_{\mathbf{obliv}}$) are sufficient to prove PMTO. For programs which are PMTO but temporarily violate uniformity, probabilistic program verification [Barthe et al. 2018] provides a mechanism for discharging proof obligations (of uniformity) introduced by unsafe casts.

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

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

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

7 CONCLUSIONS

This paper has presented $\lambda_{\mathbf{obliv}}$, 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, $\lambda_{\mathbf{obliv}}$ can be used to express probabilistic algorithms whose security depends crucially on the use of randomness. To do so, $\lambda_{\mathbf{obliv}}$ 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 $\lambda_{\mathbf{obliv}}$'s type system is powerful enough to accept sophisticated algorithms, including forms of oblivious RAMs. Such data structures were out of the reach of prior type systems. As such, our proof-of-concept implementations represent the first automated proofs that these algorithms are secure. Ultimately we hope to integrate our ideas into ObliVM [Liu et al. 2015b], a state-of-the-art compiler for oblivious algorithms, and thereby ensure that well-typed programs are indeed secure.

ACKNOWLEDGMENTS

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

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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. (2015).

Daniel E. Huang. 2017. *On Programming Languages for Probabilistic Modeling*. Ph.D. Dissertation. Harvard University.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 13. A deterministic oblivious stack built using a full (recursive) ORAM.

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

Fig. 14. A probabilistic oblivious stack built using a non-recursive ORAM. (Does not use **mux**, for simplicity.)

A APPENDIX

A.1 Case Study: Oblivious Stacks

This section considers an oblivious data structure, an *oblivious stack*. The goal of an oblivious stack is to hide both its data and which operations (pushes or pops) are taking place—only the total number of operations should be revealed. To do this, we could implement the stack using an ORAM rather than a normal (encrypted) array, and we could merge the code for push and pop so as to mask which operation is taking place (despite knowledge of the PC). Code to do this is shown in the `stackop` function in Figure 13.

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

While this code works perfectly well a *probabilistic* version of the stack, using a *non-recursive ORAM* would be more space-efficient. In particular, it will require only $O(L)$ extra space where L is the current size of the stack, whereas this version requires $O(N)$ extra space, where N is the size of the ORAM. To see how, consider that we always access a stack via its head, using the root index. Thus, in the code in Figure 13, the full `oram` internally only ever uses one slot in its position map. Thus we can do better by using an NORAM directly, having the stack manage the position tag of the root. In short, we implement an oblivious stack as a triple comprising a NORAM, the index of

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

Fig. 15. A probabilistic oblivious stack built using a non-recursive ORAM using **mux**.)

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

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

This version of an oblivious stack performs better than the version from Figure 13. The space overhead, due to the added pointers at the root and within the ORAM, is $O(L)$ where L is the size of the current stack, not the size N of the ORAM. The running time is still $O(\log N)$. Obliviousness is a direct corollary of implementing our stack on $\lambda_{\textbf{obliv}}$: Because we have labeled the stack's contents and root as secret, as well as the choice of operation, nothing can be learned about any of them when observing the event trace.

A.2 The Limits of Syntactic Uniformity Enforcement

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

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

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

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

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

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

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

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

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

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

Using this simple extension we can insert a single castNU prior to the **mux** in the eviction procedure. We can then introduce a castU at the point when the position tag is read from the ORAM on line 12 of stackop in Figure 15.

B FULL DEFINITIONS FOR PMTO PROOF

$$\overset{\bullet}{v} \in \overset{\bullet}{\text{value}} ::= v \mid \bullet \qquad \overset{\bullet}{e} \in \overset{\bullet}{\text{exp}} ::= e \mid \bullet \qquad \overset{\bullet}{t} \in \overset{\bullet}{\text{trace}} ::= \epsilon \mid \overset{\bullet}{t} \cdot \overset{\bullet}{e}$$

$$\boxed{\text{obs} \in (\text{exp} \to \overset{\bullet}{\text{exp}}) \uplus (\text{trace} \to \overset{\bullet}{\text{trace}})}$$

$$\begin{aligned}
\text{obs}(\textbf{bitv}_\text{P}(b)) &:= \textbf{bitv}_\text{P}(b) \\
\text{obs}(\textbf{bitv}_\text{S}(b)) &:= \bullet \\
\text{obs}(\textbf{flipv}(b)) &:= \bullet \\
\text{obs}(x) &:= x \\
\text{obs}(b_\text{P}) &:= b_\text{P} \\
\text{obs}(b_\text{S}) &:= \bullet \\
\text{obs}(\textbf{flip}^\rho) &:= \textbf{flip}^\rho \\
\text{obs}(\textbf{cast}_\ell(e)) &:= \textbf{cast}_\ell(\text{obs}(e))
\end{aligned}
\qquad
\begin{aligned}
\text{obs}(\textbf{mux}(e_1, e_2, e_3)) &:= \textbf{mux}(\text{obs}(e_1), \text{obs}(e_2), \text{obs}(e_3)) \\
\text{obs}(\textbf{xor}(e_1, e_2)) &:= \textbf{xor}(\text{obs}(e_1), \text{obs}(e_2)) \\
\text{obs}(\textbf{if}(e_1)\{e_2\}\{e_3\}) &:= \textbf{if}(\text{obs}(e_1))\{\text{obs}(e_2)\}\{\text{obs}(e_3)\} \\
\text{obs}(\langle e_1, e_2 \rangle) &:= \langle \text{obs}(e_1), \text{obs}(e_2) \rangle \\
\text{obs}(\textbf{let } x = e_1 \textbf{ in } e_2) &:= \textbf{let } x = \text{obs}(e_1) \textbf{ in } \text{obs}(e_2) \\
\text{obs}(\textbf{let } x_1, x_2 = e_1 \textbf{ in } e_2) &:= \textbf{let } x_1, x_2 = \text{obs}(e_1) \textbf{ in } \text{obs}(e_2) \\
\text{obs}(\textbf{fun}_y(x : \tau).\ e) &:= \textbf{fun}_y(x : \tau).\ \text{obs}(e) \\
\text{obs}(e_1(e_2)) &:= \text{obs}(e_1)(\text{obs}(e_2))
\end{aligned}$$

$$\widetilde{\text{obs}} \in \mathcal{D}(\text{trace}) \to \mathcal{D}(\overset{\bullet}{\text{trace}}) \qquad \widetilde{\text{obs}}(\widetilde{t}) := \lambda \overset{\bullet}{t}.\ \sum_t \begin{cases} \Pr\left[\widetilde{t} \overset{?}{=} t\right] & \text{if} \quad \text{obs}(t) = \overset{\bullet}{t} \\ 0 & \text{if} \quad \text{obs}(t) \neq \overset{\bullet}{t} \end{cases}$$

Fig. 16. Adversary observability (full definition)

$$
\begin{aligned}
\pi \in \mathrm{path} ::= {} & \square \\
\mid {} & \mathbf{fun}_y(x:\tau).\ \pi \\
\mid {} & \mathbf{cast}_\ell(\pi) \\
\mid {} & \mathbf{mux}(\pi,\bullet,\bullet) \mid \mathbf{mux}(\bullet,\pi,\bullet) \mid \mathbf{mux}(\bullet,\bullet,\pi) \\
\mid {} & \mathbf{xor}(\pi,\bullet) \mid \mathbf{xor}(\bullet,\pi) \\
\mid {} & \mathbf{if}(\pi)\{\bullet\}\{\bullet\} \mid \mathbf{if}(\bullet)\{\pi\}\{\bullet\} \mid \mathbf{if}_\ell(\bullet)\{\bullet\}\{\pi\} \\
\mid {} & \langle \pi,\bullet\rangle \mid \langle \bullet,\pi\rangle \\
\mid {} & \mathbf{let}\ x = \pi\ \mathbf{in}\ \bullet \mid \mathbf{let}\ x = \bullet\ \mathbf{in}\ \pi \\
\mid {} & \mathbf{let}\ x,y = \pi\ \mathbf{in}\ \bullet \mid \mathbf{let}\ x,y = \bullet\ \mathbf{in}\ \pi \\
\mid {} & \pi(\bullet) \mid \bullet(\pi)
\end{aligned}
$$

$$\boxed{\cdot@\cdot \in \underline{\mathrm{exp}} \times \mathrm{path} \rightharpoonup \underline{\mathrm{exp}}}$$

$$
\begin{aligned}
\underline{e}@\square &\triangleq \underline{e} \\
(\mathbf{fun}_y(x:\tau).\ \underline{e})@(\mathbf{fun}_y(x:\tau).\ \pi) &\triangleq \underline{e}@\pi \\
\mathbf{cast}(\underline{e})@\mathbf{cast}(\pi) &\triangleq \underline{e}@\pi \\
\mathbf{mux}(\underline{e}_1,\underline{e}_2,\underline{e}_3)@\mathbf{mux}(\pi,\bullet,\bullet) &\triangleq \underline{e}_1@\pi \\
\mathbf{mux}(\underline{e}_1,\underline{e}_2,\underline{e}_3)@\mathbf{mux}(\bullet,\pi,\bullet) &\triangleq \underline{e}_2@\pi \\
\mathbf{mux}(\underline{e}_1,\underline{e}_2,\underline{e}_3)@\mathbf{mux}(\bullet,\bullet,\pi) &\triangleq \underline{e}_3@\pi \\
\mathbf{xor}(\underline{e}_1,\underline{e}_2)@\mathbf{xor}(\pi,\bullet) &\triangleq \underline{e}_1@\pi \\
\mathbf{xor}(\underline{e}_1,\underline{e}_2)@\mathbf{xor}(\bullet,\pi) &\triangleq \underline{e}_2@\pi \\
\mathbf{if}(\underline{e}_1)\{\underline{e}_2\}\{\underline{e}_3\}@\mathbf{if}(\pi)\{\bullet\}\{\bullet\} &\triangleq \underline{e}_1@\pi \\
\mathbf{if}(\underline{e}_1)\{\underline{e}_2\}\{\underline{e}_3\}@\mathbf{if}(\bullet)\{\pi\}\{\bullet\} &\triangleq \underline{e}_2@\pi \\
\mathbf{if}(\underline{e}_1)\{\underline{e}_2\}\{\underline{e}_3\}@\mathbf{if}(\bullet)\{\bullet\}\{\pi\} &\triangleq \underline{e}_3@\pi \\
\langle \underline{e}_1,\underline{e}_2\rangle@\langle \pi,\bullet\rangle &\triangleq \underline{e}_1@\pi \\
\langle \underline{e}_1,\underline{e}_2\rangle@\langle \bullet,\pi\rangle &\triangleq \underline{e}_2@\pi \\
(\mathbf{let}\ x = \underline{e}_1\ \mathbf{in}\ \underline{e}_2)@(\mathbf{let}\ x = \pi\ \mathbf{in}\ \bullet) &\triangleq \underline{e}_1@\pi \\
(\mathbf{let}\ x = \underline{e}_1\ \mathbf{in}\ \underline{e}_2)@(\mathbf{let}\ x = \bullet\ \mathbf{in}\ \pi) &\triangleq \underline{e}_2@\pi \\
(\mathbf{let}\ x,y = \underline{e}_1\ \mathbf{in}\ \underline{e}_2)@(\mathbf{let}\ x,y = \pi\ \mathbf{in}\ \bullet) &\triangleq \underline{e}_1@\pi \\
(\mathbf{let}\ x,y = \underline{e}_1\ \mathbf{in}\ \underline{e}_2)@(\mathbf{let}\ x,y = \bullet\ \mathbf{in}\ \pi) &\triangleq \underline{e}_2@\pi \\
\underline{e}_1(\underline{e}_2)@\pi(\bullet) &\triangleq \underline{e}_1@\pi \\
\underline{e}_1(\underline{e}_2)@\bullet(\pi) &\triangleq \underline{e}_2@\pi
\end{aligned}
$$

WellPartitioned

$$\frac{\forall \pi_1 \neq \pi_2.\ e@\pi_1 = \mathbf{flipv}(\widehat{b}_1) \wedge e@\pi_2 = \mathbf{flipv}(\widehat{b}_2) \implies \left[\widehat{b}_1 \perp\!\!\!\perp \widehat{b}_2 \;\middle|\; \Phi\right]}{\Phi \vdash \underline{e}\ \mathrm{wp}}$$

$$\boxed{\Phi \vdash \underline{e}\ \mathrm{wp}}$$

Trace-Empty

$$\frac{}{\Psi, \Phi \vdash \epsilon : \tau}$$

Trace-NonEmpty

$$\frac{\Psi, \Phi \vdash \underline{t} : \tau \qquad \Phi' \vdash \underline{e}\ \mathrm{wp} \qquad \exists \Phi' \subseteq \Phi.\, \Psi, \Phi', \varnothing \vdash \underline{e} : \tau\ ;\ \varnothing}{\Psi, \Phi \vdash \underline{t}\cdot\underline{e} : \tau\ \mathrm{wp}}$$

Trace-Dist

$$\frac{\forall \underline{t}.\ \Pr\left[\widetilde{\underline{t}} \stackrel{?}{=} \underline{t}\right] > 0 \implies \exists \Psi, \Phi.\ \Psi, \Phi \vdash \underline{t} : \tau\ \mathrm{wp}}{\vdash \widetilde{\underline{t}} : \tau\ \mathrm{wp}}$$

$$\boxed{\Psi, \Phi \vdash \underline{t} : \tau\ \mathrm{wp}} \qquad \boxed{\vdash \widetilde{\underline{t}} : \tau\ \mathrm{wp}}$$

Fig. 17. The Well-partitioning Property Enforced by Affinity (full definition)

$$\boxed{\widehat{\mathcal{U}} \in \underline{\text{exp}} \to \mathcal{I}(\text{exp})}$$

$$\begin{aligned}
\widehat{\mathcal{U}}(\mathbf{fun}_y(x:\tau).\ \underline{e}) &\triangleq \lambda \overline{b}.\ \mathbf{fun}_y(x:\tau).\ \widehat{\mathcal{U}}(\underline{e})(\overline{b}) \\
\widehat{\mathcal{U}}(\mathbf{bitv_P}(b)) &\triangleq \lambda \overline{b}.\ \mathbf{bitv_P}(b) \\
\widehat{\mathcal{U}}(\mathbf{bitv_S}(\widehat{b})) &\triangleq \lambda \overline{b}.\ \mathbf{bitv_S}(\widehat{b}(\overline{b})) \\
\widehat{\mathcal{U}}(\mathbf{flipv}(\widehat{b})) &\triangleq \lambda \overline{b}.\ \mathbf{flipv}(\widehat{b}(\overline{b})) \\
\widehat{\mathcal{U}}(x) &\triangleq \lambda \overline{b}.\ x \\
\widehat{\mathcal{U}}(b_\ell) &\triangleq \lambda \overline{b}.\ b_\ell \\
\widehat{\mathcal{U}}(\mathbf{flip}^{\{\rho\}\cup R}()) &\triangleq \lambda \overline{b}.\ \mathbf{flip}^{\{\rho\}\cup R}() \\
\widehat{\mathcal{U}}(\mathbf{cast}_\ell(e)) &\triangleq \lambda \overline{b}.\ \mathbf{cast}_\ell(\widehat{\mathcal{U}}(e)(\overline{b})) \\
\widehat{\mathcal{U}}(\mathbf{mux}(\underline{e}_1,\underline{e}_2,\underline{e}_3)) &\coloneqq \lambda \overline{b}.\ \mathbf{mux}(\widehat{\mathcal{U}}(\underline{e}_1)(\overline{b}),\widehat{\mathcal{U}}(\underline{e}_2)(\overline{b}),\widehat{\mathcal{U}}(\underline{e}_3)(\overline{b})) \\
\widehat{\mathcal{U}}(\mathbf{xor}(\underline{e}_1,\underline{e}_2)) &\coloneqq \lambda \overline{b}.\ \mathbf{xor}(\widehat{\mathcal{U}}(\underline{e}_1)(\overline{b}),\widehat{\mathcal{U}}(\underline{e}_2)(\overline{b})) \\
\widehat{\mathcal{U}}(\mathbf{if}(\underline{e}_1)\{\underline{e}_2\}\{\underline{e}_3\}) &\coloneqq \lambda \overline{b}.\ \mathbf{if}(\widehat{\mathcal{U}}(\underline{e}_1)(\overline{b}))\{\widehat{\mathcal{U}}(\underline{e}_2)(\overline{b})\}\{\widehat{\mathcal{U}}(\underline{e}_3)(\overline{b})\} \\
\widehat{\mathcal{U}}(\langle \underline{e}_1,\underline{e}_2\rangle) &\coloneqq \lambda \overline{b}.\ \langle \widehat{\mathcal{U}}(\underline{e}_1)(\overline{b}),\widehat{\mathcal{U}}(\underline{e}_2)(\overline{b})\rangle \\
\widehat{\mathcal{U}}(\mathbf{let}\ x = \underline{e}_1\ \mathbf{in}\ \underline{e}_2) &\coloneqq \lambda \overline{b}.\ \mathbf{let}\ x = \widehat{\mathcal{U}}(\underline{e}_1)(\overline{b})\ \mathbf{in}\ \widehat{\mathcal{U}}(\underline{e}_2)(\overline{b}) \\
\widehat{\mathcal{U}}(\mathbf{let}\ x,y = \underline{e}_1\ \mathbf{in}\ \underline{e}_2) &\coloneqq \lambda \overline{b}.\ \mathbf{let}\ x,y = \widehat{\mathcal{U}}(\underline{e}_1)(\overline{b})\ \mathbf{in}\ \widehat{\mathcal{U}}(\underline{e}_2)(\overline{b}) \\
\widehat{\mathcal{U}}(\underline{e}_1(\underline{e}_2)) &\coloneqq \lambda \overline{b}.\ \widehat{\mathcal{U}}(\underline{e}_1)(\overline{b})(\widehat{\mathcal{U}}(\underline{e}_2)(\overline{b}))
\end{aligned}$$

$$\boxed{\widehat{\mathcal{U}} \in \underline{\text{trace}} \to \mathcal{I}(\text{trace})}$$

$$\begin{aligned}
\widehat{\mathcal{U}}(\epsilon) &\triangleq \lambda \overline{b}.\ \epsilon \\
\widehat{\mathcal{U}}(\underline{t}\cdot\underline{e}) &\triangleq \lambda \overline{b}.\ \widehat{\mathcal{U}}(\underline{t})(\overline{b})\cdot\widehat{\mathcal{U}}(\underline{e})(\overline{b})
\end{aligned}$$

$$\boxed{\widetilde{\mathcal{U}}^* \in \mathcal{D}(\underline{\text{trace}}) \to \mathcal{D}(\text{trace})}$$

$$\begin{aligned}
\widetilde{\mathcal{U}}(\underline{t}) &\triangleq \lambda t.\ \Pr\left[\widehat{\mathcal{U}}(\underline{t}) \stackrel{?}{=} t\right] \\
\widetilde{\mathcal{U}}^*(\widetilde{t}) &\triangleq \lambda t.\ \sum_{\underline{t}} \Pr\left[\widetilde{t} \stackrel{?}{=} \underline{t}\right] \Pr\left[\widetilde{\mathcal{U}}(\underline{t}) \stackrel{?}{=} t\right]
\end{aligned}$$

Fig. 18. Mixed Semantics Simulation (full definition)

$\boxed{\underline{e} \sim \underline{e}}$

REFL
$$\frac{}{\underline{e} \sim \underline{e}}$$

SECRET-LIT
$$\frac{}{b_S \sim b'_S}$$

SECRET-BITV
$$\frac{}{\mathbf{bitv}_S(\widehat{b}) \sim \mathbf{bitv}_S(\widehat{b'})}$$

SECRET-FLIPV
$$\frac{}{\mathbf{flipv}(\widehat{b}) \sim \mathbf{flipv}(\widehat{b'})}$$

CONG-1
$$\frac{\underline{e} \sim \underline{e'}}{\begin{array}{c}\mathbf{fun}_y(x:\tau).\ \underline{e} \sim \mathbf{fun}_y(x:\tau).\ \underline{e'} \\ \mathbf{cast}_\ell(\underline{e}) \sim \mathbf{cast}(\underline{e'})\end{array}}$$

CONG-2
$$\frac{\underline{e}_1 \sim \underline{e}'_1 \qquad \underline{e}_2 \sim \underline{e}'_2}{\begin{array}{c}\mathbf{xor}(\underline{e}_1, \underline{e}_2) \sim \mathbf{xor}(\underline{e}'_1, \underline{e}'_2) \\ \langle \underline{e}_1, \underline{e}_2 \rangle \sim \langle \underline{e}'_1, \underline{e}'_2 \rangle \\ \mathbf{let}\ x = \underline{e}_1\ \mathbf{in}\ \underline{e}_2 \sim \mathbf{let}\ x = \underline{e}'_1\ \mathbf{in}\ \underline{e}'_2 \\ \mathbf{let}\ x_1, x_2 = \underline{e}_1\ \mathbf{in}\ \underline{e}_2 \sim \mathbf{let}\ x_1, x_2 = \underline{e}'_1\ \mathbf{in}\ \underline{e}'_2 \\ \underline{e}_1(\underline{e}_2) \sim \underline{e}'_1(\underline{e}'_2)\end{array}}$$

CONG-3
$$\frac{\underline{e}_1 \sim \underline{e}'_1 \qquad \underline{e}_2 \sim \underline{e}'_2 \qquad \underline{e}_3 \sim \underline{e}'_3}{\begin{array}{c}\mathbf{mux}(\underline{e}_1, \underline{e}_2, \underline{e}_3) \sim \mathbf{mux}(\underline{e}'_1, \underline{e}'_2, \underline{e}'_3) \\ \mathbf{if}(\underline{e}_1)\{\underline{e}_2\}\{\underline{e}_3\} \sim \mathbf{mux}(\underline{e}'_1)\{\underline{e}'_2\}\{\underline{e}'_3\}\end{array}}$$

$\boxed{\widetilde{\underline{t}} \sim \widetilde{\underline{t}}}$

TRACE-EMPTY
$$\frac{}{\epsilon \sim \epsilon}$$

TRACE-NONEMPTY
$$\frac{\underline{t} \sim \underline{t'} \qquad \underline{e} \sim \underline{e'}}{\underline{t}\cdot\underline{e} \sim \underline{t'}\cdot\underline{e'}}$$

$\boxed{\underline{t} \sim \underline{t}}$

TRACE-DIST
$$\frac{\forall \underline{t}, q > 0.\ \Pr\left[\widetilde{\underline{t}} \stackrel{?}{=} \underline{t}\right] = q \Rightarrow \exists \underline{t'}.\ \Pr\left[\widetilde{\underline{t'}} \stackrel{?}{=} \underline{t'}\right] = q \wedge \underline{t} \sim \underline{t'}}{\widetilde{\underline{t}} \sim \widetilde{\underline{t'}}}$$

Fig. 19. Low-equivalence for Mixed Terms (full definition)