ABSTRACT

Title of Dissertation: LANGUAGE-BASED TECHNIQUES FOR SECURE PROGRAMMING

Ian Sweet
Doctor of Philosophy, 2022

Dissertation Directed by: Professor Michael Hicks
Department of Computer Science

Secure Computation (SC) encompasses many different cryptographic techniques for computing over encrypted data. In particular, *Secure Multiparty Computation* [179, 72] enables multiple parties to jointly compute a function over their secret inputs. MPC languages offer programmers a familiar environment in which to express their programs, but fall short when confronted with problems that require *flexible coordination*. More broadly, SC languages do not protect non-expert programmers from violating *obliviousness* or expected *bounds on information leakage*. We aim to show that secure programming can be made safer through language-based techniques for expressive, coordinated MPC; probabilistically oblivious execution; and quantitative analysis of information flow. We begin by presenting Symphony, an expressive MPC language that provides flexible coordination of many parties, which has been used to implement the secure shuffle of Laur, Willemson, and Zhang. Next, we present $\lambda_{\mathbf{Obliv}}$, a core language guaranteeing that well-typed programs are probabilistically oblivious, which has been used to type check tree-based, nonrecursive ORAM (NORAM). Finally, we present a novel application of dynamic analysis techniques to an existing system for enforcing bounds on information leakage, providing a better balance of precision and performance.

LANGUAGE-BASED TECHNIQUES FOR SECURE PROGRAMMING

by

Ian Sweet

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2022

Advisory Committee:
Professor Michael Hicks, Chair/Advisor
Professor Lawrence Washington, Dean's Representative
Professor David Darais
Professor Jonathan Katz
Professor David Van Horn

Acknowledgements

To my wonderful family – Allyn, Steve, Ellen, Skyler, Abbey, and Aidan – I cannot put into words how influential you have been. So much of who I am has been shaped by you. Thank you for all your love and support. I love you.

My wife, Camille, has assured me that she will never read these acknowledgements. I considered calling her bluff, but here I am acknowledging her anyway. Camille and I met in high school, dated all through college, and were married smack dab in the middle of my PhD. I have never regretted a second I've spent with her. I could go on, but she said I "better not write some long thing" about her.

My advisor, Mike Hicks, deserves all the thanks I can possibly give and then some. All of the most important lessons and skills I learned during my PhD came directly from him. He taught me to value and cultivate my community. He taught me the importance of communication. He taught me how to conduct honest, meaningful research. What strikes me most as I write this, though, is that he always treated me as an equal. I hope one day I am able to mentor someone else and make them feel the way Mike made me feel.

I would also like to acknowledge my committee – David Darais, David Van Horn, Jon Katz, and Larry Washington – for their help shaping my proposal and dissertation. David Van Horn's enthusiasm for programming languages inspired me to pursue graduate studies. David Darais' research agenda and aesthetic has helped shape my own. Jon Katz's questions and suggestions pushed my research to be truly

interdisciplinary.

I am indebted to David Heath and Daniel Noble for helping me understand a research area in which I had no prior experience. I've benefited immeasurably from their patience and kindness. I am indebted to Ethan Cecchetti for helping me understand a research area in which I had substantial prior experience. I regret that our paths didn't cross sooner. I have a deep appreciation for Ethan's generosity of spirit. Whether it is his time or his tea, he gives freely and with enthusiasm.

Finally, I must give my sincerest thanks to the dear friends that I made along the way. José Calderón has been a consistent source of joy and support from the very beginning. When I was struggling, José helped remind me what is important. I mean it sincerely when I say I could not have done it without him. Kesha Hietala, Sankha Guria, and James Parker are my academic siblings. They will never be rid of me, whether they like it or not. My experience was infinitely richer because they were a part of it.

Table of contents

Chapter 1

Introduction

The field of *Secure Computation* encompasses different cryptographic techniques for computing over encrypted data [67, 74, 35, 42, 179, 72]. For example, *Fully Homomorphic Encryption* (FHE) [67] can be used to outsource a computation over secret data to an untrusted server and *Zero-Knowledge Proofs* (ZKP) [74] can be used to prove properties about secret data from an untrusted client [32]. Of particular importance to this dissertation is *Secure Multiparty Computation* [179, 72] which enables multiple parties to jointly compute a function over their secret data.

Problem: Lack of Expressiveness Secure computation based on MPC stands out due to its generality and efficiency. MPC technology today is hundreds of millions of times faster than technology from a mere 20 years ago, meaning that many applications have become feasible. Modern MPC languages and frameworks are also increasingly convenient, offering programmers a familiar language with which to express their MPC programs [176, 5, 63, 181, 125, 155, 36]. Unfortunately, while these frameworks allow the programmer to effectively express many MPCs, they fall short when confronted with problems that require *flexible coordination*.

Overwhelmingly, MPC frameworks take the default view that all parties perform the same synchronized activity, in the style of single-instruction multiple-data

(SIMD). The SIMD view of computation is not always appropriate. In many scenarios, it is useful if the parties can each execute *different* computations. For example, suppose we wish to implement a round-based card game where N players use MPC to jointly shuffle and deal the cards. By using MPC, the parties ensure that the deck and the players' individual hands remain secret. At the same time, each party might choose to play its cards according to her own strategy, and so each party might carry out different actions, perhaps by interacting with a user via I/O. Moreover, a party might drop out of the computation altogether once eliminated from the game. As another example, suppose that a very large number of parties wish to provide inputs to a privacy-preserving computation. In such situations, it is pragmatic for the parties to elect a small committee to carry out the computation on their behalf. Doing this can greatly aid efficiency, since the performance of most MPC primitives degrades as the number of parties grows.

To implement these kinds of applications, the programmer must carefully coordinate the parties. Unfortunately, most existing MPC frameworks offer no coordination features; non-synchrony is handled by ad hoc mechanisms, or not at all. Ad hoc mechanisms can lead to programming mistakes, and these mistakes can result in (potentially random) hangs or wrong answers. The one prior MPC language that does provide coordination support, Wysteria [143], lacks *expressiveness* and *ergonomics*. For example, individual parties may not delegate computations over their inputs to other parties, and MPCs must be expressed in a rigid sublanguage that makes interleaving encrypted and plaintext computation awkward and inefficient.

Problem: Lack of Obliviousness When programming in languages for secure computation we often want to write programs that correspond closely to the programs we would we write in a less exotic language (e.g. C or Java). Unfortunately, secure computation requires that programs be expressible in a circuit model rather than

the more familiar RAM-model. A naive approach to solving this problem is simply to declassify secret values and then use them as usual. Of course, this is not safe as it can lead to side-channel vulnerabilities through observation of memory access patterns [91, 114, 189] and instruction timing [37, 95] (as made famous by recent Spectre and Meltdown attacks [97, 107, 171]).

One way to mitigate side-channel vulnerabilities is to store secret values in *oblivious RAM* (ORAM) [170, 114]. First proposed by Goldreich [71] and Goldreich and Ostrovsky [73], 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 [108, 111, 114] 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, Hicks, and Shi [108] and Liu et al. [111, 109] developed a family of type systems to check when *partial* use of ORAM (alongside normal, encrypted RAM) results in no loss of security; i.e., only when the addresses of secret data could indirectly reveal sensitive information must the data be stored in ORAM. This optimization can provide order-of-magnitude asymptotic performance improvements. Wang et al. [174] explored how to build *oblivious data structures* (ODSs), such as queues or stacks, that are more efficient than their standard counterparts implemented on top of ORAM. In follow-up work, Liu et al. [112] devised ObliVM, a programming language for implementing such oblivious data structures, including ORAMs themselves. A key feature of ObliVM is careful treatment of random values, 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.

Problem: Uncontrolled Information Release A secure computation guarantees that execution will not leak any information about secret inputs *beyond what is released by the result.* It is the responsibility of users to audit the program and ensure that the result will not release "too much" information about their secret. Consider a simple case in which two parties, **A** and **B**, have secret values, a and b respectively. There are a number of scenarios in which **A** and **B** may want to perform secure computation:

- **A** is allowed to query a sensor network, controlled by **B**, with sensitive features such as the location of personnel in a building.
- **A** is allowed to query a database, controlled by **B**, which contains sensitive information such as electronic health records.
- **A** and **B** are nations that want to determine if they have military units within a particular range of each other without revealing their exact position.

In highly sensitive situations like these, **A** and **B** would like some assurance that the function is not releasing more information than they expect. Unfortunately, existing languages for secure computation do not allow the programmer to specify or enforce bounds on the amount of information released by the result.

Controlled information release can be achieved by *quantifying* how much information a function, f, releases about the secret input a (resp. b). Intuitively, **A** is interested in how uncertain **B** is about a after observing $f(a, b)$. The amount of information released by $f(a, b)$ can be measured as the change in **B**'s uncertainty about a. For example, if $f(a, b) = a \bmod 2^b$ and **B** observes $f(a, 1) = 0$ then **B**'s uncertainty is cut in half.

The intuition about uncertainty is formalized by modeling **B**'s uncertainty as a probability distribution, δ, over the possible values of a [163, 44]. The information release is quantified as the change in the *Bayes Vulnerability* of δ, which is the

probability that **B** can guess a in a single try: $V(\delta) \stackrel{\text{def}}{=} \max_a \delta(a)$. In other words, information release measures how much more likely **B** is to guess a in a single try after observing $f(a, b)$. Various works [115, 28, 78, 103] propose rejecting f if there exists a result that causes the vulnerability to exceed a fixed threshold K. In particular, if the vulnerability $V(\delta)$ exceeds K after observing $f(a, b)$ *for any possible value of* a then a should refuse to execute the program f.

Mardziel et al. [115] propose a *sound* analysis technique for automatically quantifying information release according to vulnerability using abstract interpretation [47]. In particular, they approximate uncertainty, δ, using an abstract domain called *probabilistic polyhedron* (PP), which pair standard numeric abstract domain, such as *convex polyhedra* [48], with some additional *ornaments*, which include lower and upper bounds on the size of the support of the distribution, and bounds on the probability of each possible secret value. Using PP can yield a precise, yet safe, estimate of the vulnerability. Unfortunately, PPs can be very inefficient. Defining *intervals* [46] as the PP's numeric domain can dramatically improve performance, but only with an unacceptable loss of precision. Can we retain the precision afforded by convex polyhedra while also enjoying the performance of intervals?

A promising approach to addressing the issues above is the use of techniques from programming languages research, including type systems and dynamic analysis.

Thesis Statement Secure programming can be made safer through language-based techniques for expressive, coordinated MPC; probabilistically oblivious execution; and quantitative analysis of information flow.

1.1 A Language for Expressive, Coordinated Secure Multiparty Computation

Our first contribution is the development of Symphony, an expressive MPC language that provides appropriate tools for coordinating large numbers of parties. Symphony, presented in Chapter 3, draws inspiration from prior work on coordinated MPC languages [143] and on *choreographic languages* [50, 51, 137, 124]. This combination can improve the flexibility and expressiveness of MPC languages without sacrificing coordination and deadlock-freedom.

Symphony's most interesting language features are:

- *Scoped parallel expression blocks*, or *par* blocks, allow the programmer to easily control which parties execute which code. The programmer annotates a *par* block with the (dynamically computed) set of parties who should enter the block. The language ensures that parties executing within the block agree on the logical values of local variables, so there is no risk of deadlocks or undefined behavior. Crucially, Symphony retains a generalized SIMD character: We prove that the developer can reason about and debug her program as if it runs sequentially, even though in practice the program is executed in a distributed deployment.

- *First-class shares* model multiparty-encrypted values in a way that allow the programmer to freely *delegate* computation from one party set to another, and to *reactively* mix MPC operations with cleartext operations.

These features have previously appeared, in part, in other MPC frameworks: Our *par* blocks are inspired by Wysteria, and our first-class shares are inspired by the programming style of EMP [176] and Obliv-C [181]. Symphony is the first MPC language to carefully *combine* these features, and to *generalize* them, e.g., by allowing *re-sharing*

of already encrypted values among a different set of parties. This combination of features is crucial for flexible coordination: by allowing the programmer to mix first-class shares and *par* blocks, we ensure that she can write arbitrarily complex MPCs where parties freely enter, leave, and shift the locus of cryptographic computations.

Contributions Our specific contributions are as follows.

- We motivate the need for Symphony, describe its key features using examples, and compare it to the state of the art (Section 3.1, Section 3.7).
- We present a core formalism for Symphony, called $\lambda_{\mathbf{MPC}}$—syntax (Section 3.2.1), *single-threaded* semantics (Section 3.2.2), and *distributed* semantics (Section 3.3). We prove that the single-threaded semantics faithfully represents the distributed semantics, and thus can be used as the basis for reasoning about a Symphony program's behavior (Section 3.4).
- We describe our prototype implementation for Symphony, which leverages EMP [176] and MOTION [36] as its cryptographic backends (Section 5.6).
- We discuss the 16 programs we have implemented in Symphony, showing that it provides *ergnomic* and *expressiveness* advantages over Wysteria, its closest competitor: Symphony programs tend to be shorter and more direct, and support optimizations involving delegation and resharing that Wysteria cannot (Section 3.6.1). We present a complete implementation of the LWZ secure shuffling protocol [105], highly useful in a variety of scenarios [79, 60], that no prior MPC language can express (Section 3.1.2).
- We show that despite its expressiveness, Symphony enjoys good performance. On a set of kernel benchmarks running on a simulated LAN, we show that Symphony takes a mean 1.15× the time taken by Obliv-C, a state-of-the-art MPC implementation for C (Section 3.6.2).

SYMPHONY is publicly available at https://github.com/plum-umd/symphony-lang.

1.2 A Language for Probabilistically Oblivious Computation

Our second contribution is the development of $\lambda_{\textbf{Obliv}}$, a core language for oblivious computation, inspired by ObliVM. $\lambda_{\textbf{Obliv}}$, presented in Chapter 4, extends a standard language with primitives for securely generating and using uniformly distributed random numbers. We prove that $\lambda_{\textbf{Obliv}}$'s type system guarantees *probabilistic memory trace obliviousness* (PMTO), i.e., that the distribution of adversary-visible execution traces is independent of secret values. This property generalizes the deterministic MTO property enforced by Liu, Hicks, and Shi [108] and Liu et al. [111], which did not consider the use of randomness. In carrying out this work, we discovered that the ObliVM type system is unsound, so an important contribution of $\lambda_{\textbf{Obliv}}$ is a design which achieves soundness without overly restricting or complicating the language.

$\lambda_{\textbf{Obliv}}$'s type system aims to ensure that no probabilistic correlation forms between secrets and publicly revealed random choices. The language provides a mechanism for transitioning secret random values to public ones—which we call a *revelation*—which 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 ensures that revelations are uniformly distributed by treating randomly generated numbers as *affine*, meaning they cannot be freely copied. Affinity prevents revealing the same number twice, which is problematic because a second revelation is not uniformly distributed when conditioned on observing the first. Unfortunately, strict affinity is too strong for implementing oblivious algorithms, which

require the ability to make copies of random numbers which are later revealed. $\lambda_{\mathbf{Obliv}}$'s type system addresses this by allowing random numbers to be copied and then ensuring that they can never be revealed. Moreover, $\lambda_{\mathbf{Obliv}}$ ensures that random numbers do not themselves influence whether or not they are revealed, as this could also result in a non-uniform revelation. The type system prevents this behavior 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.) We prove that $\lambda_{\mathbf{Obliv}}$ enjoys PMTO by relating its semantics to a novel *mixed semantics* whose terms operate on distributions directly, which makes stating and proving the PMTO property much easier.

$\lambda_{\mathbf{Obliv}}$ is expressive enough to type check interesting algorithms. We present the implementation of a tree-based, non-recursive ORAM (NORAM) that type checks in a straightforward extension of $\lambda_{\mathbf{Obliv}}$; we have implemented a type checker for this extension. NORAM is a key component of state-of-the-art ORAM implementations [160, 169, 175] and other oblivious data structures [174], and to our knowledge ours is the first implementation automatically verified to be oblivious. We additionally show that *recursive* ORAM, built on NORAM, is possible but requires a few more advanced (but standard) language features we have not implemented, including region polymorphism, recursive and variant types, and existential quantification.

Finally, we have also experimented with implementing oblivious data structures using our NORAM implementation. We conclude by providing evidence that *oblivious stacks* (ostacks) don't satisfy PMTO due to the possibility of information leakage caused by overflow in the underlying NORAM. Instead, ostacks satisfy statistic security as long as the underlying NORAM has a negligible probability of overflow with respect to its size.

Contributions Our specific contributions are as follows.

- We motivate the need for $\lambda_{\textbf{Obliv}}$, describe its key features using examples, and show that ObliVM's type system is not PMTO (Section 5.1).
- We present the formal syntax, semantics, and type system of $\lambda_{\textbf{Obliv}}$ (Section 5.2) before proving that well-typed $\lambda_{\textbf{Obliv}}$ programs satisfy PMTO (Section 4.3).
- We describe our prototype typechecker for OblivML, an extension of $\lambda_{\textbf{Obliv}}$ with standard features, and provide the first implementation of NORAM that has been automatically verified as oblivious (Section 4.4).
- We show that oblivious stacks, a particular form of ODS [174], do not satisfy PMTO due to the possibility of information leakage caused by overflow in the underlying NORAM.

OblivML is publicly available at https://github.com/plum-umd/oblivml.

1.3 Refining Probabilistic Bounds on Information Leakage

Our third contribution is a new approach that ensures a better balance of precision and performance for dynamic enforcement of knowledge-based security policies [115]. Our approach, presented in Chapter 5, augments PP with two new techniques which can be used to improve their precision. Both techniques begin by analyzing a program, f, using intervals as the underlying numeric abstract domain for the PP.

Our first technique is to use *sampling* to augment the result. We execute the program using the possible secret values i sampled from the posterior δ' derived from a particular output o_i. If the analysis were perfectly accurate, executing $f(i)$ would produce o_i. But since intervals are overapproximate, sometimes it will not. With many sampled outcomes, we can construct a Beta distribution to estimate the size of

the support of the posterior, up to some level of confidence. We can use this estimate to boost the lower bound of the abstraction, and thus improve the precision of the estimated vulnerability.

Our second technique is of a similar flavor, but uses symbolic reasoning to magnify the impact of a successful sample. We once again execute a sample which is consistent with the posterior distribution but this time we do so *concolically* [157], thus maintaining a symbolic formula (called the *path condition*) that characterizes the set of variable valuations that would cause execution to follow the observed path. We then count the number of possible solutions and use the count to boost the lower bound of the support (with 100% confidence).

Finally, sampling and concolic execution can be combined for even greater precision.

Contributions Our specific contributions are as follows.

- We have formalized and proved our techniques are sound (Section 5.2, Section 5.4, Section 5.5) with respect to the dynamic enforcement technique of Mardziel et al. [115].

- We have implemented and evaluated our approaches (Section 5.6, Section 5.7), using a privacy-sensitive ship planning scenario (Section 5.1). We find that our techniques provide similar precision to convex polyhedra while providing orders-of-magnitude better performance.

Our implementation is publicly available at `https://github.com/GaloisInc/TAMBA`.

Chapter 2

Background

2.1 Secure Multiparty Computation

Secure Multiparty Computation (MPC) is a subfield of cryptography that allows mutually untrusting parties to compute arbitrary functions of their private inputs while revealing only the function output. That is, MPC allows parties to work together to run programs *under encryption*.

For example, consider a scenario in which two parties, Alice and Bob, want to compute who is richer without revealing their net worth to each other. They can use an MPC protocol that implements the program $a > b$ where a and b are Alice and Bob's wealth respectively. An MPC protocol allows Alice and Bob to compute $a > b$ and guarantees that they learn no more than if a trusted third party had performed the comparison on their behalf. This scenario, originally posed by Yao [178], is known as the *Millionaire's Problem*.

The Millionaire's Problem illustrates a helpful mental model for understanding MPC. An MPC protocol can be understood as a cryptographic technique for simulating a trusted third party when no such party is available.

2.1.1 Protocol Characterization

An MPC protocol can be characterized according to four criteria: number of supported parties, semantics, security model, and round complexity. Next, we discuss these four criteria and summarize some representative MPC protocols accordingly.

Semantics The semantics of a protocol determines the sort of values that may be encrypted and what operations are allowed on them. Typically, a distinction is drawn between boolean (binary) and arithmetic protocols. Boolean protocols encrypt booleans (bits), and permit XOR and AND operations. Arithmetic protocols encrypt values according a finite algebraic ring or field and permit addition and multiplication operations.

Security Model The security model of a protocol explicates assumptions about the adversary. We assume that the adversary uses a static corruption strategy, meaning that they must choose which parties they wish to corrupt before the protocol begins executing. Next, we'll discuss the adversary's behavior, power, and corruption threshold. The adversary's behavior can be either *semi-honest* (i.e. passive) or *malicious* (i.e. active)[1] A semi-honest adversary doesn't deviate from the protocol, so they can only learn secrets by observing values that are communicated to the parties they have corrupted. In contrast, a malicious adversary may deviate from the protocol arbitrarily.

The adversary's power is either *bounded* or *unbounded*. A bounded adversary can only invest polynomial time in κ[2] into trying to learn secrets. An unbounded adversary may invest an arbitrary amount of time. Almost all MPC protocols, especially those which are efficient in practice, rely on computationally secure cryptographic schemes (e.g. El-Gamal encryption [65]).

[1] We ignore covert adversaries, as they are less common.

[2] κ is a computational security parameter defined by the MPC protocol.

Table 2.1: Representative MPC protocols and their properties. Categorizes protocols according to the number of supported parties, semantics, security model (`S,M` for semi-honest and malicious, `B,U` for bounded and unbounded, `D,H` for dishonest and honest majorities) and round complexity.

Protocol	**Parties**	**Semantics**	**Security**	**Rounds**
Yao [179]	2	$\mathbb{B}$	`S,B,D`	Constant
GMW [72]	N	$\mathbb{R}$	`S,B,D`	Depth
BGW [27]	N	$\mathbb{F}$	`S,U,H`	Depth
BMR [24]	N	$\mathbb{B}$	`S,B,D`	Constant
BaMaRo [13]	2	$\mathbb{Z}_m$	`S,B,D`	Constant
TinyOT [131]	2	$\mathbb{B}$	`M,B,D`	Depth
WRK [177]	2	$\mathbb{B}$	`M,B,D`	Constant
SPDZ [52]	N	$\mathbb{F}$	`M,B,D`	Depth

Finally, the adversary's corruption threshold bounds the number of parties they may corrupt. In an *honest majority* protocol, the adversary cannot corrupt more than $\frac{N}{2} - 1$ out of N parties. In a *dishonest majority* protocol, the adversary can corrupt $N - 1$ out of N parties. Honest majority protocols are generally more efficient than dishonest majority protocols, and are required to achieve security against unbounded adversaries [27].

Round Complexity The round complexity of a protocol describes how many rounds of bidirectional communication it must perform to execute a circuit, C, in terms of its multiplicative depth $|C|$.[3] Only the multiplicative depth is considered because most modern protocols are able to execute addition gates without any communication [98, 72]. Protocols have *constant* round complexity when the number of communication rounds is independent of $|C|$. Protocols have *depth* round complexity when the number of communication rounds is proportional to $|C|$.

Table 2.1 provides a summary of some representative MPC protocols and their properties. For a more comprehensive overview of MPC, see Lindell [106].

[3] This footnote is an explicit acknowledgement that I am abusing notation.

2.1.2 Protocol Descriptions

The two main approaches to MPC protocols are *garbled circuits* [179, 24], and *secret sharing* [72, 27]. We describe each of these approaches shortly and discuss some of their advantages and disadvantages. Before we can do so, we must shine a light on a core cryptographic primitive used in almost every MPC protocol.

Oblivious Transfer Behind every great MPC protocol there is a great cryptographic primitive called *Oblivious Transfer* [138] (OT). A $\binom{2}{1}$[4] OT [59] enables a party **A** (the "sender") to send 1 out of 2 possible messages to **B** (the "receiver") without **A** learning which message they chose. Unfortunately, Impagliazzo and Rudich [89] proved that OT requires public-key cryptography. This is concerning because public-key encryption is much more expensive than public key encryption. Luckily, Beaver [26] showed that a small number of so-called "base" OTs can be extended into many more OTs using only private key encryption. The first efficienct construction of OT extension was given by Ishai et al. [90]. Modern MPC protocols typically do not use $\binom{2}{1}$ OT directly, instead relying on variations such as Random and Correlated OT [59]. Some modern MPC protocols, such as SPDZ [52], rely instead on *Fully Homomorphic Encryption* [67] (FHE). We will not remark much on FHE, except to say that it is cheap in terms of communication but very expensive in terms of computation.

Garbled Circuits The garbled circuits approach to MPC, originally due to Yao [179], enables two parties, **A** (the "garbler") and **B** (the "evaluator"), to compute a boolean circuit C. The garbling approach works by having **A** encrypt the circuit C and send it to **B** who will then evaluate it.

A encrypts the wires by generating two κ-bit (a typical value is $\kappa = 128$) labels, logically representing 0 and 1. **A** encrypts the gates by representing them as truth

[4] Another blatant abuse of notation, someone stop this mad man!

tables, replacing the values of the truth table with the corresponding labels (determined by the input wires), and encrypting the output label in each row using the input labels in that row as the key. After encrypting the truth table, she must also "garble" the gate by randomly permuting the rows.

A sends both the encrypted circuit and the labels for her input wires over to **B**. At this point, all **B** needs to evaluate the circuit are the labels corresponding to his inputs. **A** sends the appropriate labels to **B** using $\binom{2}{1}$ OT. The OT ensures that **A** does not observe which label is chosen by **B**. At this point, **B** can evaluate the circuit. He evaluates a gate by decrypting each of the four rows using the labels on the input wires as the key, determining the label of the output wire according to which decryption succeeds. After **B** is done evaluating, the result can be revealed to **A**, **B**, or both. The result is revealed to **A** by having **B** send her the labels of the output wires. The result is revealed to **B** by having **A** send him the mapping between logical values and output labels.

Yao's protocol has constant round complexity because it only uses OT to handle **B**'s input to the protocol. The Free-XOR optimization [98] allows XOR gates to be computed without any symmetric key encryption. Furthermore, **A** no longer needs to encrypt or send XOR gates to **B**. The Point-And-Permute [24] optimization allows **B** to decrypt the correct row on the first try. This reduces the number of symmetric key decryptions that **B** must execute for each gate from 4 to 1. The Half-Gate [182] optimization reduces the number of rows in an encrypted AND gate from 4 to 2, reducing the communication cost of sending the encrypted circuit. The BMR [24] protocol generalizes the 2-party protocol to N parties by simulating the garbler using an N party MPC protocol (e.g. based on secret sharing). The protocol retains constant round complexity by garbling the gates in parallel.

Secret Sharing The additive secret sharing approach to MPC, originally due to Goldreich, Micali, and Wigderson [72], is analogous to Yao's protocol in that it enables two parties, **A** and **B**, to compute a boolean circuit C. The secret sharing approach works by having **A** and **B** exchange secret shares of their input and then execute the circuit C on their shares in parallel.

A splits her input a into two *secret shares* σ and $\sigma \oplus a$, where σ is a uniform random bit. **B** does the same for his input b. We denote **A**'s share of a (b) as $[a]_A$ ($[b]_A$) and **B**'s share of a (b) as $[a]_B$ ($[b]_B$). The shares are secret because neither σ nor $\sigma \oplus a$ reveal any information about a to **B**. The shares are shares because the original value can be reconstructed by XOR'ing the shares together: $[a]_A \oplus [a]_B = \sigma \oplus (\sigma \oplus a) = a$.

To compute C, **A** and **B** homomorphically compute XOR and AND operations over their shares. Computing XOR gates turns out to be easy. Each party computes their share of the XOR simply by taking the XOR of their shares.

$$\begin{aligned} a \oplus b &= ([a]_A \oplus [a]_B) \oplus ([b]_A \oplus [b]_B) \\ &= ([a]_A \oplus [b]_A) \oplus ([a]_B \oplus [b]_B) \end{aligned}$$

Unfortunately, AND gates cannot be computed locally because there are cross-terms $[a]_A \wedge [b]_B$ and $[a]_B \wedge [b]_A$ involving shares of both parties.

$$\begin{aligned} a \wedge b &= ([a]_A \oplus [a]_B) \wedge ([b]_A \oplus [b]_B) \\ &= [a]_A \wedge [b]_A \oplus [a]_A \wedge [b]_B \oplus [a]_B \wedge [b]_A \oplus [a]_B \wedge [b]_B \end{aligned}$$

So, to compute an AND gate, **A** chooses a random bit σ and uses it as her share of $a \wedge b$: $[a \wedge b]_A = \sigma$. Then, **A** sends **B** his share using $\binom{4}{1}$ OT with **B** choosing the share corresponding to his values of $[a]_B$ and $[b]_B$: $[a \wedge b]_B = \sigma \oplus ([a]_A \oplus [a]_B) \wedge ([b]_A \oplus [b]_B)$.

GMW has depth round complexity because each AND gate requires OT. The complexity scales with the depth of the circuit, rather than the size, because all the

AND gates at the same depth of the circuit can be executed in parallel. The GMW protocol can be generalized naturally to N parties without issue. Furthermore, the GMW approach can also be generalized to arithmetic circuits by using precomputed multiplication triples [25] to implement AND gates instead of OT. Instead of additative secret shares, th BGW [27] protocol uses threshold secret shares [158] in the honest majority setting.

2.2 Information Flow

Having discussed how to allow secret data to be computed over securely, it seems only natural that we now discuss how to prevent it from being computed over insecurely.

Information flow control (IFC) systems provide provably correct, automatic enforcement of security properties. Intuitively, IFC applies concepts from the literature on access control to data manipulated by a program. For example, IFC has been used to ensure that web applications which store sensitive information in an SQL database don't accidentally leak that information publicly [134, 168].

In this dissertation we advocate for language-based approaches to IFC, such as type systems are static analyses. Language-based approaches are necessary because the security of a program is intimately tied to its semantics. For example, the following Rust function yields `0` if boolean disjunction is evaluated left to right and short-circuited but yields `x` otherwise.

```
fn f (x: u32) -> u32 {
  let mut ret = 0;
  let _ = true || { ret = x; false };
  ret
}
```

IFC Characterization An IFC system typically guarantees confidentiality, integrity, or both. A *confidentiality* property prevents sensitive information from being read, and an *integrity* property prevents it from being written (i.e. modified). In this dissertation, we only consider confidentiality properties.[5] Likewise, IFC systems can be characterized as either dynamic or static. A *dynamic* IFC system enforces the security property by monitoring a program as it executes. In contrast, a *static* IFC system enforces the security property by analyzing the program prior to execution. In this dissertation, we only consider static IFC systems like type systems and static analyses.

Information Flow Lattices Denning [55] observed that the security requirements of a system can be specified according a lattice structure. The elements of the lattice are security labels which specify the sensitivity level of data. For example, a simple set of labels are public (P) and secret (S). The partial order over these elements specifies how data is allowed to flow through the program. For example, the ordering $P \sqsubset S$ indicates that public data may flow into a secret context, but not vice versa. Finally, the join operation is required to compute the labels of data which is derived from data which is already labeled. For example, the result of $x + y$ where x is labeled P and y is labeled S can be computed as $P \sqcup S = S$. We will often describe things in terms of public and secret labels, leaving implicit the understanding that we may substitute an arbitrary security lattice without issue.

Noninterference Goguen and Meseguer [70] proposed noninterference as a general, rigorous way to define security properties. The definition presupposes that we have some way of labeling data, with modern presentations using security lattices as described above. Intuitively, a program satisfies noninterference if changing secret

[5] As observed by Biba [29], however, there is a duality between confidentiality and integrity which can often be fruitfully exploited [45].

inputs has no observable effect on public outputs. With information flow lattices and noninterference we are able to precisely and rigorously specify what it means for a program to be secure. However, we have not yet considered any ways in which noninterference might be enforced. Next, we consider an enforcement mechanism for IFC based on static type systems.

Security-Typed Languages Building on the work of Denning [55] and Goguen and Meseguer [70], Volpano, Irvine, and Smith [172] present a static type system which provably ensures that programs satisfy noninterference. A language of this kind is typically called a security-typed language, by virtue of the fact that well-typed programs provably enjoy security (as defined by noninterference). There has been a tremendous amount of work since on the metatheory [81, 1, 34] and application [128] of security-typed languages. An important limitation of these approaches is that they define noninterference only in terms of the *extensional* (i.e. input-output) behavior of the program.

Oblivious Languages When performing IFC on programs that implement cryptography, it is particularly important to consider the *intensional* (e.g. timing, memory traces) behaviors of the program. In the context of security, intensional behaviors of a program are known as *side-channels* and are indirectly observable in practice. For example, attacks have been demonstrated through observation of memory access patterns [91, 114, 189] and instruction timing [37, 95] (as made famous by recent Spectre and Meltdown attacks [97, 107, 171]). The feasibility of these attacks suggests that applying IFC to cryptography requires a maximally intensional variant of noninterference, called obliviousness. A program is *oblivious* [108] if changing the secret inputs has no observable effect on the execution trace. Put another way, a program is oblivious if it is noninterfering when the intensional aspects of a program are considered observable. Recent work has demonstrated that security-typed languages

can be extended to enforce obliviousness [108, 110, 111].

Declassification and Quantitative Information Flow Noninterference is often not precise enough to define the security of programs in the wild. Noninterference is an example of a security policy that is *qualitative* because it only allows us to specify where information is allowed to flow. In contrast, a *quantitative* policy is one that allows us to specify not just where information may flow, but how much. To see why this might be useful, consider the following two Rust functions where the input `x` is considered secret and the output is considered public.

```
fn id   (x: u32) -> u32  { x }
fn eq73 (x: u32) -> bool { x == 73 }
```

A qualitative security policy will disallow both `id` and `eq73`, since both propagate information about `x` to the output. However, a quantitative security policy could choose to allow `eq73` but disallow `id` based on the observation that information leaked by `eq73` is small.

Clarkson, Myers, and Schneider [44] advocates defining quantative information flow according to how an attacker's belief changes as a result of observing the execution of a program. The attacker's *belief* is a probability distribution over the possible values of the secret. The probability assigned to a value corresponds to its likelihood from the attacker's perspective. The belief is updated, according to Bayesian inference, whenever they observe the execution of the program on some input. The information leakage is defined as the change in entropy of the attacker's belief.

Mardziel et al. [115] provide an enforcement mechanism for quantified information flow, according to the approach described by Clarkson, Myers, and Schneider [44]. They use *abstract interpretation* [47], a form of static analysis, to compute a sound approximation of the attacker's belief. This is used to bound the informaton leakage by measuring the maximum change in entropy over all possible inputs. The result of

this analysis is used dynamically to determine whether or not to execute the program, according to a specified bound on information leakage. If the program is executed, the approximation of the attacker's belief is revised according to the actual input they provide.

Chapter 3

λ_{MPC}: A Language for Expressive, Coordinated Secure Multiparty Computation

This chapter presents Symphony, an expressive MPC language that provides appropriate tools for coordinating large numbers of parties. The most interesting language features of Symphony are *scoped parallel expressions*, or `par` blocks, and *first-class shares*. Scoped parallel expressions allow the programmer to easily control which parties execute which code. First-class shares model multiparty-encrypted values in a way that allow the programmer to freely *delegate* computation from one party set to another, and to *reactively* mix MPC operations with cleartext operations.

Importantly, in the style of choreographic programming languages [50, 51, 137, 124], Symphony ensures that programs are deadlock-free by construction. This is what allows Symphony to retains a generalized SIMD character, even though in practice the program is executed using a distributed deployment. To prove that Symphony programs are deadlock-free, we relate a sequential semantics that cannot encounter deadlock with a distributed semantics that can. We prove a soundness

theorem that ensures that programs which do not encounter errors when executed sequentially will not encounter errors when deployed according to the distributed semantics.

In Section 3.1 we discuss the advantages of SYMPHONY over existing MPC languages, and introduce SYMPHONY by showing an implementation of an MPC protocol from the literature. Then, in Section 3.2 we formalize the syntax and semantics of $\lambda_{\mathbf{MPC}}$ which is a minimal core language that captures the essential features of SYMPHONY. We then present the distributed semantics $\lambda_{\mathbf{MPC}}$ in Section 3.3 before stating and proving sequential soundness theorem in Section 3.3. In Section 5.6 we describe our implementation of SYMPHONY before evaluating its expressiveness and performance in Section 3.6.

3.1 Overview

Most MPC frameworks express secure computation via an extension to a familiar programming language [176, 5, 63, 181, 125, 155]. The frameworks leverage standard language features such as numeric types, loops, and arrays, but distinguish conceptually *encrypted* values, which are part of the MPC and not visible to the participating parties, from *cleartext* ones. For example, here is "millionaires" in Obliv-C [181], in which two parties, A and B, which to learn which is richer without revealing their total wealth:

```
1  void millionaire(void* args) {
2    protocolIO* io = args;
3    obliv int v1 = feedOblivInt(io->mywealth, A);
4    obliv int v2 = feedOblivInt(io->mywealth, B);
5    obliv bool ge = v1 >= v2;
6    revealOblivBool(&io->cmp, ge, 0);
7  }
```

Both parties run this program, SIMD-style. Variables `v1` and `v2` are encrypted, as per the `obliv` keyword. Function `feedOblivInt` sets the initial values of these variables. On party A, line 3 encrypts the input stored in A's copy of `io->mywealth`, while line 4 does likewise for B. Internally, the function will `send` its encrypted input to the other party, which synchronously `recv`s it; i.e., line 3 sends from A to B and line 4 from B to A. Line 5 computes on these encrypted values (at both parties), with the result itself being encrypted. Finally, `revealOblivBool` coordinates among the two parties to decrypt the result; it is stored in each party's `&io->cmp`.

Under the hood, Obliv-C uses *garbled circuits* [179] to carry out these operations; other frameworks use *secret shares* [72, 27] or *homomorphic encryption* [67]. Regardless, most provide a similar SIMD-style programming model, either as library calls within an existing language (EMP [176], MPyC [155], MOTION [36]), or as a direct extension of that language (PICCO [188], ObliVM [112], and SCALE-MAMBA [5]). Sharemind [142] has developed its own C-like language, SecreC, which compiled to Sharemind assembly. Other works, such as CBMC-GC [63] and Frigate [125], compile subsets of C into circuits.

3.1.1 Problem: Coordination

Suppose we wish to write a program in which not all parties do the same thing. In Obliv-C you could write the following to execute `<code>` *only* at party X

```
if (ocCurrentParty()== X){ <code> }
```

Other Obliv-C constructs also take party identifiers, e.g., `readInt` reads from local storage on the identified party. Using such constructs requires care. Suppose A wishes to share the encrypted result of computing `f` on its input `a`. The following code to do so contains a bug:

```
1  int a;
```

```
2   readInt(&a, "input.txt", A);
3   obliv int share;
4   if (ocCurrentParty() == A) {
5     share = feedOblivInt(f(a), A);
6   }
7   ... // proceed with secure computation on f(a)
```

Due to the conditional on line 4, the share on line 5 will trigger a `send` on A but no corresponding `recv` on B, causing A to block forever awaiting B's response. We fix the issue by dropping the conditional on line 4, so `feedOblivInt` is called at both parties.

```
1   int a;
2   readInt(&a, ..., A);
3   obliv int share = feedOblivInt(f(a), A);
```

The code no longer hangs on A, but now there is another problem: undefined behavior. The call to `readInt` on line 2 initializes `a` on A but not on B. The call to `f(a)` causes both A and B to read the value contained in `a`, causing undefined behavior on B. Our final solution is to provide a dummy value for `a` when executing on B.[1]

```
1   int a;
2   readInt(&a, ..., A);
3   int fa = ocCurrentParty() == A ? f(a) : 0;
4   obliv int share = feedOblivInt(fa, A);
```

There is still a risk: `f` must not perform any communication among parties, otherwise we will experience another coordination error like the one in the first example.

Experienced MPC programmers may be able to avoid such pitfalls assuming coordination requirements do not get too complex. Unfortunately, complexity is more likely when writing MPCs for $N > 2$ parties. In general, we might have dozens or

[1] We could have instead ensured that `a` is initialized to a dummy value on B. This works, but when dealing with compound types (e.g. an array of integers) this requires allocating memory on B for all of A's input and initializing the memory with dummy values.

more parties, with interactions between overlapping sets of parties. Each party's role may shift over time, possibly dependent on prior computations. These complexities are perhaps the reason that, with the exception of Wysteria [143, 144], N-party languages like PICCO [188], Frigate [125], and SCALE-MAMBA [5] do not provide coordination mechanisms. Wysteria still suffers serious limitations, which we consider in detail in Section 3.6.

3.1.2 Symphony: Expressive, Coordinated MPC

Symphony is a new MPC programming language that supports coordinated MPC for $N \geq 2$ parties by generalizing the standard SIMD-style view. It is a dynamically typed functional programming language with support for integers, pairs, variant (sum) types, lists, let-binding, pattern matching, (recursive) higher-order functions, and (mutable) references and arrays.

We illustrate Symphony by showing how to use it to implement an efficient *Secure Shuffle* protocol: Each party in a set P provides an array of inputs, and the protocol concatenates and shuffles them in encrypted form so that the parties do not know the origin of each element. The shuffled inputs can then be sorted efficiently because knowing the result of comparisons between shuffled elements tells nothing about the original inputs. Sorting is useful for follow-on algorithms, such as binary search.

LWZ Laur, Willemson, and Zhang [105] showed how to implement a highly efficient secure shuffle among parties Q which is resilient to T corruptions using a linear secret. The LWZ protocol implements a secure shuffle by repeatedly shuffling the elements among committees (i.e., strict subsets of the parties) of size $|Q| - T$. The committee is given shares of the input list and agrees on a random permutation, π; locally permutes their shares according to π; and then constructs new shares for the next committee. This process repeats until each set of T parties has been excluded from

```
party A B C D E

-- read input at p, secret-share to all in Q
def readShare Q p = par ({ p } \/ Q)
  let i = par { p } read (array int) from "lwz.txt" in -- file local to each p
  share [gmw, array int : { p } -> Q] i

def delegateShares P Q =
  map (readShare Q) (psetToList P)

def shuffleWith Q S sharesQ = par (Q \/ S)
  let sharesS = share [gmw, array int : Q -> S] sharesQ in
  share [gmw, array int : S -> Q] (shuffle S sharesS)

def lwz Q sharesQ =
  let t = 1 in
  foldr (shuffleWith Q) shares (subsets Q ((psetSize Q) - t))

def revealLte Q x y = reveal [gmw, bool : Q -> Q] x <= y

def secureSort Q sharesList =
  let sharesQ  = par Q (arrayConcat sharesP) in
  let shuffled = lwz Q sharesQ in
  let sorted   = quickSort (revealLte Q) shuffled

def main () = par {A,B,C,D,E}
  let Q = {A,B,C} in
  let sharesList = delegateShares {A,B,C,D,E} Q in
  let sorted     = secureSort Q sharesList in
  ...
```

Figure 3.1: A secure, N-party sorting procedure written in SYMPHONY. Uses the Shuffle-Then-Sort paradigm [79] with LWZ as the underlying shuffle. Each party in $\{A, B, C, D, E\}$ contributes an array of integers, which are concatenated together and then securely shuffled and sorted by the parties in Q.

some committee, which is sufficient to hide the global shuffle. Figure 3.1 lists LWZ in Symphony. To the best of our knowledge, no other MPC language can express this protocol, due to its coordination challenges. We explain SYMPHONY's features as we explain its LWZ implementation.

As is standard, all parties run the same program, starting at `main`. While the shuffling and sorting code works for arbitrary numbers of parties, `main` is specialized to those parties declared at the top, named A–E.

Par blocks SYMPHONY uses *par blocks* to permit some, but not all, parties to execute a part of the program. For example, when `par {A,B,C,D,E} ...` is reached on line 26, only the listed parties execute the subsequent code block. Parties not in the given set skip the block, returning an *opaque value* ★ which will cause an error if computed upon (since no real value is available at that party).

Located data As SYMPHONY programs are fundamentally distributed, data is *located* at particular parties' hosts. An important invariant is that the same logical variable will be bound to the same logical value on each executing party within a `par`. As a result, those parties are naturally coordinated when operating on that data, avoiding errors that can arise when coordination is more ad hoc, e.g., using conditional execution based on party ID. SYMPHONY provides an abstraction called a *bundle* to allow different parties to use the same variable to hold different values; these can be viewed as maps from party ID to value. We don't use these in LWZ.

First-class party sets Party sets like `{A,B,C,D,E}` are run-time values in SYMPHONY, not static annotations. On line 27, `Q` is bound to the set `{A,B,C}` and is then passed as the second argument to `delegateShares` on line 28, and as the first to `secureSort` on line 29, which in turn provides it to `par` on line 22. First-class party sets allow protocols to be generic in the number of parties, and the located-data invariant allows the choice of parties to be reliably coordinated at run-time.

First-class Shares and Delegation Encrypted values in SYMPHONY are called *shares*, which we can think of as secret shares among a particular set of parties. (SYMPHONY implements both GMW and Yao back ends.) We use `share`$[\phi, \tau : P \to Q]$ v to take value v now at P and secret-share it among parties Q, where ϕ is the MPC protocol (e.g., `gmw`) and τ is the type of the share's contents. Oftentimes P is a single party and Q is a set. For example, in `readShare Q p`, party `p` reads an array

of integers from local file `lwz.txt` (line 5), and then creates a share among parties in `Q` (in SYMPHONY a share of an array is an array of shares). The `share` operation requires all parties in $\{p\} \cup Q$ to be present (ensured by the `par` on line 4) so that `p` can transmit to each party in `Q` its share and know they are ready to receive it—note that `p` may or may not be a member of `Q`. The `delegateShares` function calls `readShare Q` for each party `p` $\in$ `P`, with the goal of *delegating* the subsequent computation to those parties in `Q`.

Re-sharing Calling `share` on an existing share will *reshare* it. This is what happens on lines 12–13: shares among parties `Q` are reshared to be among parties `S`, and then reshared back to `Q` once shuffled by `S` (more below). Language support for resharing is unique to SYMPHONY, and it is critical for implementing LWZ, which we can now explain. The `lwz` function securely shuffles `sharesQ`, which are shares among parties `Q`. The `foldr` on line 17 invokes `shuffleWith` on each size $|Q| - T$ subset `S` of `Q` (computed by `subsets`). In turn, this function reshares `sharesQ` among those parties in `S`, which invoke `shuffle` to permute its values, and then reshare the result back. Within `shuffle`, the parties `S` agree on a seed for a PRNG that they use as the basis for the shuffle, ensuring they compute the same permutation. Laur, Willemson, and Zhang proved that if each subset `S` has size $|P| - T$, then nothing can be learned about the order of the shuffled elements unless $n > T$ parties collude. In Figure 3.1, we specify $T = 1$ on line 16.

Revelation and Reactive MPC Once line 23 completes, `shuffled` contains a shuffled, secret-shared array. Line 24 then `quicksort`s this array, using `revealLte Q` as the comparison function. This function uses SYMPHONY's `reveal` construct—while `share` converts its argument at P to shares at Q, `reveal` does the reverse, converting shares at P to plaintext at Q. Here, `reveal` is used to perform *reactive MPC*: each comparison is revealed to plaintext, allowing the bulk of the sorting function to occur

locally, at each party. This makes sorting much faster than performing it "within" an MPC, as computations on shares directly, but no less secure because we shuffled the elements first. Once sorted, we could perform other operations on the array, e.g., a binary search, with similar privacy benefits.

3.2 $\lambda_{\mathbf{MPC}}$: Syntax and Semantics

$\lambda_{\mathbf{MPC}}$ is a minimal core language which captures the essential features of Symphony. This section presents its syntax and *single-threaded* (ST) semantics.

3.2.1 Syntax

The syntax of $\lambda_{\mathbf{MPC}}$ is shown in Fig. 3.2. To simplify the formal semantics, the syntax adheres to a kind of *administrative normal form* (ANF), meaning that most expression forms operate directly on variables x, rather than subexpressions e, as is the case in the actual implementation.[2] We isolate *atomic expressions* a as a subcategory of full expressions e; the former evaluate to a final result in one "small" step.

Most of the syntactic forms are standard. Binary operations apply either to integers or shares (e.g., $+$, $\times$) or to party sets (e.g., $\cup$). Conditionals $x \mathbin{?} x \diamond x$ correspond to multiplexor expressions, i.e., `mux if` in Symphony. Pairs are accessed via projection (e.g., $\pi_1\ \langle 1, 2\rangle$ evaluates to 1), while sums (aka *variants* or *tagged unions*) are accessed via pattern matching (e.g., $\texttt{case}\ \iota_1\ 0\ \{y.\ e_1\}\ \{y.\ e_2\}$ evaluates to e_1 wherein y is substituted with 0).[3] Party sets are also accessed via `case` and processed like lists—the first branch handles the $\varnothing$ case, while the second binds two variables, one for a selected party and the other for the rest of the set. Recursive functions are

[2] This restriction does not harm expressiveness. Direct-style syntax like $\texttt{ref}\ (1 + \texttt{read})$ can be encoded in $\lambda_{\mathbf{MPC}}$'s formal syntax like $\texttt{let}\ x = \texttt{read}\ \texttt{in}\ \texttt{let}\ y = 1\ \texttt{in}\ \texttt{let}\ z = y + x\ \texttt{in}\ \texttt{ref}\ z$.

[3] We can encode $\texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ as $\texttt{let}\ z = 0\ \texttt{in}\ \texttt{let}\ y = x\ ?\ \iota_1\ z \diamond \iota_2\ z\ \texttt{in}\ \texttt{case}\ y\ \{\_.\ e_1\}\ \{\_.\ e_2\}$. **TODO**: fix spacing

$i \in$	$\mathbb{Z}$			*integers*
$A, B, C \in$	party			*parties*
$m, p, q \in$	pset	$\triangleq$	$\wp(\text{party})$	*sets of parties*
$x, y, z \in$	var			*variables*
$\odot \in$	bop			*binary ops (+, ×, ∪, ...)*
$a \in$	atom	$::=$	x	*variable reference*
		$\mid$	i	*integer literal*
		$\mid$	p	*party set literal*
		$\mid$	$x \odot x$	*binary operation*
		$\mid$	$x \mathbin{?} x \diamond x$	*conditional*
		$\mid$	$\iota_i\ x$	*sum injection*
		$\mid$	$\langle x, x \rangle$	*pair creation*
		$\mid$	$\pi_i\ x$	*pair projection*
		$\mid$	$\lambda_z x.\ e$	*(rec.) function def*
		$\mid$	`ref` x	*reference creation*
		$\mid$	$!x$	*dereference*
		$\mid$	$x := x$	*reference assignment*
		$\mid$	`read`	*read int input*
		$\mid$	`write` x	*write output*
		$\mid$	`share`$[x \to x]\ x$	*share encrypted val.*
		$\mid$	`reveal`$[x \to x]\ x$	*reveal encrypted val.*
$e \in$	expr	$::=$	a	*atomic expression*
		$\mid$	`case` $x\ \{\bar{x}.e\}\{\bar{x}.e\}$	*elim for sums, psets*
		$\mid$	$x\ x$	*function call*
		$\mid$	`par` $x\ e$	*parallel execution*
		$\mid$	`let` $x = e$ `in` e	*let binding*

Figure 3.2: $\lambda_{\mathbf{MPC}}$ formal syntax.

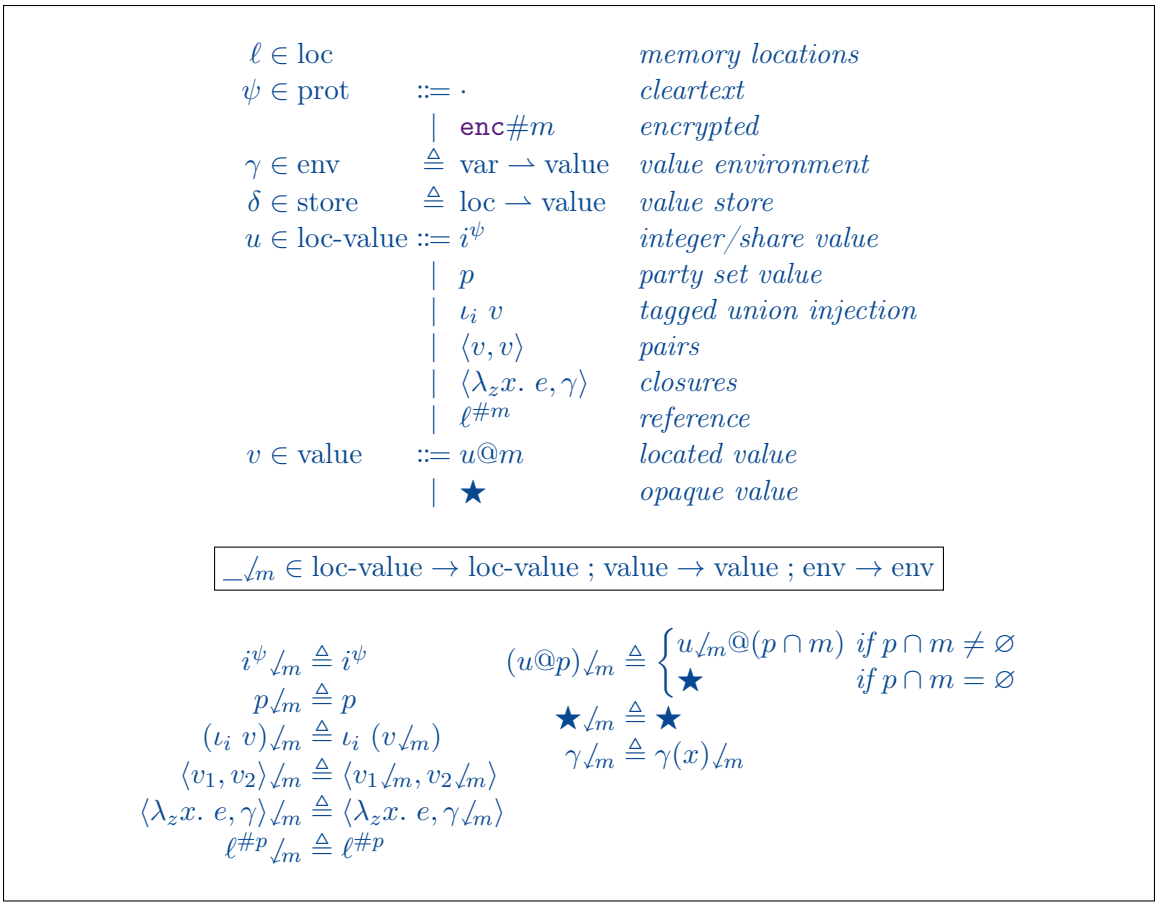


Figure 3.3: $\lambda_{\mathbf{MPC}}$ definitions and metafunctions used in formal semantics.

written $\lambda_z x.\ e$; the function body e may refer to itself via variable z. $\lambda_{\mathbf{MPC}}$ also has mutable references, and primitives for I/O. $\lambda_{\mathbf{MPC}}$ does not model lists or bundles because they are easily encoded; we explain how when discussing the semantics. The MPC-related constructs `par`, `share`, and `reveal` match their Symphony counterparts; the latter two elide the output type and protocol annotation (which are useful for an implementation but unnecessary for formal modeling). Symphony's implementation generalizes other aspects of $\lambda_{\mathbf{MPC}}$, too, as discussed in Section 5.6; e.g., it permits sharing values of any type, and doing case analysis on encrypted sums.

3.2.2 Overview

The ST semantics for $\lambda_{\mathbf{MPC}}$ models all participating parties as if they were executing in lockstep. We prove that the ST semantics faithfully models the *distributed* (DS) semantics presented in Section 3.3, according to which parties may act independently. Thus, the ST semantics can serve as the basis of $\lambda_{\mathbf{MPC}}$ formal reasoning, e.g., about correctness and security.

The main judgment $\varsigma \longrightarrow \varsigma$ is a reduction relation between *configurations* ς. A configuration is a 5-tuple comprising the current mode m, environment γ, store δ, stack κ, and expression e. The mode is the set of parties computing e in parallel; we say the parties $A \in m$ are *present* for a computation. Per Figure 3.3, environments are partial maps from variables to values, and stores are partial maps from memory locations to values; we discuss stacks shortly. The main judgment employs judgment $\gamma \vdash_m \delta, a \hookrightarrow \delta, v$, which defines the reduction of atomic expressions a to values v. The rules for both judgments are given in Figure 3.4.

3.2.3 Values

Values v have one of two forms: $u@m$ indicates that the *located value* u is only accessible to $A \in m$, e.g., because it was the result of evaluating e in mode m; whereas $\bigstar$ is the *opaque value* which is both unknown and inaccessible. Located values are defined in Figure 3.3, including for numbers i^{ψ}, sets of parties p, sums $\iota_i\ v$, pairs $\langle v, v \rangle$, recursive functions $\langle \lambda_z x.\ e, \gamma \rangle$ which include a closure environment γ, and memory locations (i.e., pointers) $\ell^{\#p}$. These are standard except for annotations ψ and $\#p$.

The annotation $\#p$ to indicate the parties p that are *co-creators* of the referenced memory, whereas ψ indicates the *protocol* of the annotated integer: $\cdot$ represents a cleartext value,[4] whereas $\texttt{enc}\#p$ represents an encrypted value shared among parties

[4]We write just i when the annotation ψ is $\cdot$.

$B \in p$ (a "share"). Thus, a value $1^{\texttt{enc}\#q}@q$ can be read as "an integer 1, encrypted (i.e., secret shared) between parties q, and accessible to parties q." The first q represents *among whom is this value shared* (determined when the share is created), and the second q represents *who has access to this value* (determined by the enclosing `par` blocks). Location annotations are only used in the ST semantics in order to simulate the presence of multiple parties; they are unused in the distributed semantics and final execution. On the other hand, the $\texttt{enc}\#q$ and $\#p$ annotations are used during distributed execution to detect buggy programs which fail to coordinate properly, e.g., if A alone attempts to do arithmetic on a share owned by both A and B, or if only A attempts to write to a reference it co-created with B.

3.2.4 Operational Semantics

Now we consider some of the operational rules.

Literals, Variables, and Binding Rule ST-Var retrieves a variable's value from the environment and *(re)locates it* to the current mode m via $\gamma(x)\downarrow_m$. The function $\_\downarrow_m$ is given in Figure 3.3. For values $u@p$, $\_\downarrow_m$ relocates them to $p \cap m$, unless the intersection is empty in which case the value is inaccessible, so it becomes $\bigstar$. Relocating is a deep operation; $u@p\downarrow_m$ also relocates the contents u to $u\downarrow_m$, which recurses over the sub-terms of u. Relocation ensures that the retrieved value is *compatible with* m. A value v is compatible with a set of parties m when it is accessible to some set of parties $p \subseteq m$. Compatibility with the current mode is a general invariant of all of the rules. Rule ST-Lit types integer and principal literals, annotating them with (compatible) location m.

Local variable bindings with `let` are managed using a stack κ, which is either the empty stack $\top$ or a list of frames $\langle \texttt{let}\ x = \square\ \texttt{in}\ e \mid m, \gamma\rangle :: \kappa$. To evaluate $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, we push frame $\langle \texttt{let}\ x = \square\ \texttt{in}\ e_2 \mid m, \gamma\rangle$ and set the active expression to

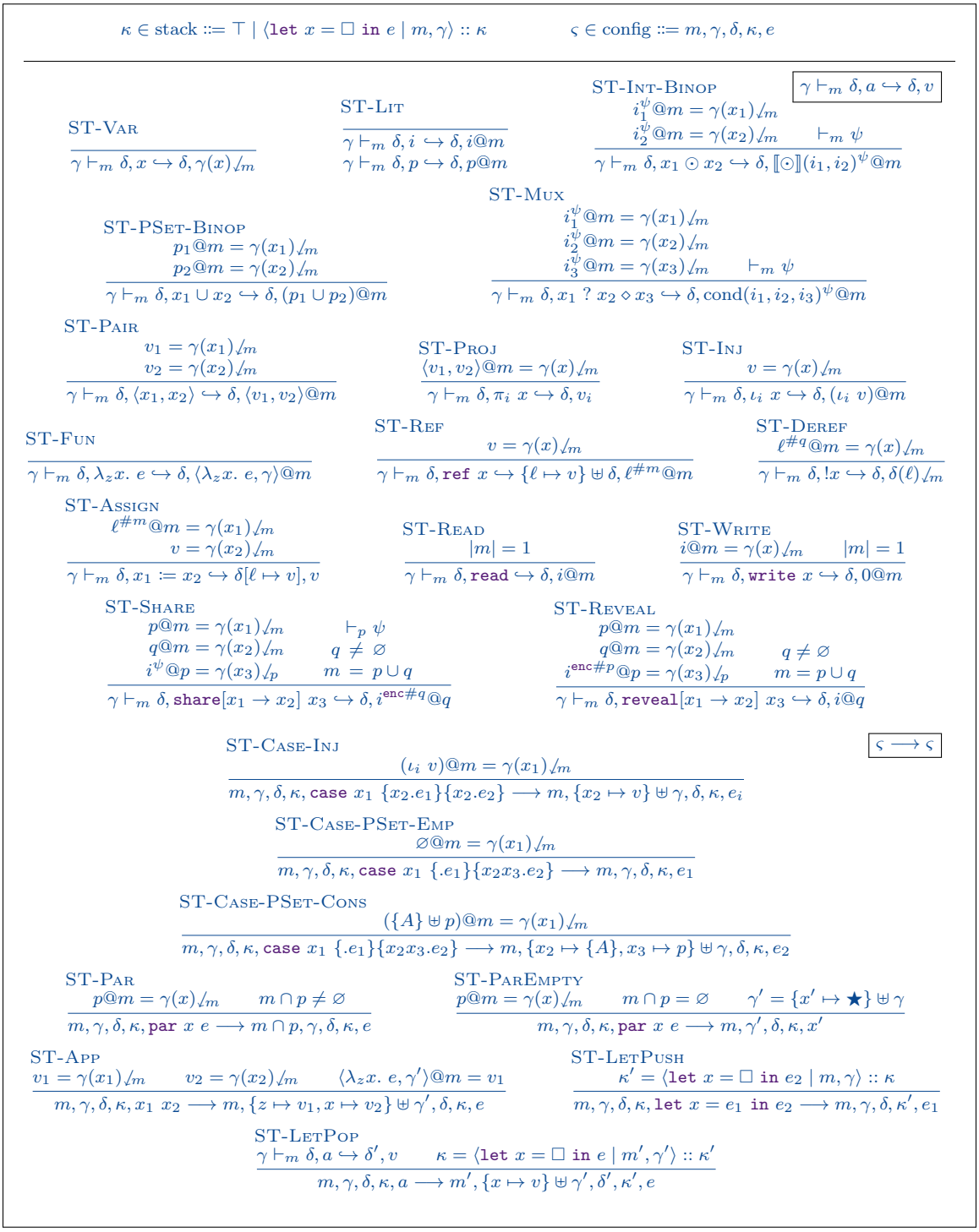


Figure 3.4: $\lambda_{\mathbf{MPC}}$ sequential semantics.

e_1 (Rule ST-Let-Push, not shown). When an expression evaluates to a value v, the topmost frame $\langle \texttt{let}\ x = \square\ \texttt{in}\ e_2 \mid m, \gamma \rangle$ is popped and evaluation proceeds on e_2 using saved mode m and environment γ, the latter updated to map x to v (Rule ST-Let-Pop, not shown).

Rule ST-Int-Binop handles arithmetic over integers. This rule illustrates another invariant that all elimination rules share. To compute on a value while running in mode m requires that the value be accessible to all parties in m. We see this in premises like $i_1^{\psi}@m = \gamma(x_1)\downarrow_m$, which locate the operated-on variable to current mode m and then ensure that the value's location is also m, i.e., all parties have access to the computed-on value. Doing so ensures that these parties, when running in a distributed setting with their own store, environment, etc. will agree on the result. For this rule in particular, we also ensure that both integers have the same protocol ψ, and that this protocol is compatible with mode m, written $\vdash_m \psi$. Compatibility holds when ψ is cleartext, and when it is $\texttt{enc}\#m$, i.e., i is a share amongst all parties currently present. Compatibility checks are used in the distributed semantics, too, to ensure parties are in sync.

Par mode Operationally, $\texttt{par}\ x\ e$ evaluates e in mode $m \cap p$ where $p@m = \gamma(x)\downarrow_m$; i.e., only those parties in p which are *also* present in m will run e. When $m \cap p$ is non-empty, rule ST-Par directs e to evaluate in the refined mode. If $m \cap p$ is empty, then per rule ST-ParEmpty, e is skipped and $\bigstar$ is returned.[5] Note that because the stack tracks each frame's mode, when the current expression completes the old mode will be restored when a stack frame is popped.

Here is an example of how par mode and variable access interact.

[5]Since $\bigstar$ is not an expression—it is a value—we return a fresh variable and the environment with that variable mapped to $\bigstar$.

$$\texttt{par}\ \{A,B\}\ \texttt{let}\ x = \texttt{par}\ \{A\}\ 1\ \texttt{in}$$
$$\texttt{let}\ y = \texttt{par}\ \{B\}\ x\ \texttt{in}$$
$$\texttt{let}\ z = \texttt{par}\ \{C\}\ 2\ \texttt{in}\ x$$

The outer $\texttt{par}\ \{A,B\}$ evaluates its body in mode $\{A,B\}$, per rule ST-Par. Next, according to rules ST-LetPush, ST-Par, and ST-Int we evaluate 1 in mode $m = \{A,B\} \cap \{A\} = \{A\}$; we bind the result $1@\{A\}$ to x in γ per rule ST-LetPop. Next, according to rules ST-LetPush, ST-Par and ST-Var we evaluate x in mode $m = \{B\}$. Per rule ST-Var, we retrieve value $1@\{A\}$ for x, and then $\not\downarrow_{\{B\}}(1@\{A\})$ yields $\bigstar$ as the result, which is bound to y in γ per rule ST-LetPop. This result makes sense: Party B reads variable x whose contents are only accessible to A, so all it can do is return the opaque value. Finally, $\texttt{par}\ \{C\}\ 2$ evaluates to $\bigstar$ according to rule T-ParEmpty, since $m = \{A,B\} \cap \{C\} = \varnothing$. This $\bigstar$ result is bound to z per rule ST-LetPop, and the final result x, evaluated in mode $m = \{A,B\}$ is $\not\downarrow_{\{A,B\}}(1@\{A\}) = 1@\{A\}$ per rule ST-Var.

Sums, Pairs, and Party Sets; Encodings Sums and pairs are essentially standard, modulo the consideration of their values' locations, and party sets are constructed via set-union, and deconstructed via pattern matching (see Section 3.2.1).

Symphony directly supports lists and arrays; in $\lambda_{\mathbf{MPC}}$ they can be encoded by iterated sum and pair values where $\texttt{nil} \triangleq \iota_1\ 0$ and $\texttt{cons} \triangleq \lambda x.\ \lambda xs.\ \iota_2\ \langle x, xs\rangle$; lists can be deconstructed by pattern matching with `case`. We can encode bundles as an *association list*, implementing a map from parties to values located at that party. For example, the following list represents a bundle with 8 located at A and 3 located at B.

$$\iota_2\ \langle\langle\{A\}, 8@\{A\}\rangle, \iota_2\ \langle\langle\{B\}, 3@\{B\}\rangle, (\iota_1\ 0)\rangle\rangle$$

(Missing location annotations for the list itself are dropped to avoid clutter; they are

all $@\{A, B\}$.)

References Rule ST-Ref creates a fresh reference in the usual way, returning a located pointer, but annotated with the parties that created it. Rule ST-Deref takes a reference located in the current mode m and returns the pointed-to contents made compatible with m. Rule ST-Assign updates the store with the new value and returns it, as usual, but only works for $\ell^{\#p}$ references where $p = m$, the current mode. Why? Consider the following example.

$$\begin{aligned}&\texttt{par}\ \{A, B\}\ \texttt{let}\ x = \texttt{ref}\ 0\ \texttt{in}\\ &\qquad\qquad\texttt{let}\ \_ = (\texttt{par}\{A\}\ x := 1)\ \texttt{in}\\ &\qquad\qquad\texttt{let}\ y = !x\ \texttt{in}\ \ldots\end{aligned}$$

The variable x initially contains a reference $\ell^{\#\{A,B\}}$ because it was created in a context with mode $m = \{A, B\}$. Then x is assigned to by A in the `par` expression on the subsequent line. By rule ST-Assign, the creators of the reference $\#\{A, B\}$ must match mode m to proceed, but since m is $\{A\}$ the program is stuck. This is desirable because to proceed would cause A's and B's views of the computation to get out of sync. When we run this program at each of A and B separately, as part of the distributed semantics, on A we would do the assignment but on B it would be skipped. As such, on A the value of y would be 1 but on B it would be 0. If the continuation of the program in $\ldots$ were to branch on y and then in one branch do some MPC constructs but not in the other, then the two parties would become even further out of sync.

I/O Rules for handling I/O (not shown) require that the mode is a singleton party; this is important for ensuring compatibility (i.e., so that all parties agree on the contents of shared variables).

Multiparty computation Party A creates encrypted values (i.e., shares) among parties q using syntax $\texttt{share}[x_1 \to x_2]\ x_3$ handled by rule ST-Share. Variable x_1 is the set of input parties p; variable x_2 is the (nonempty) set of parties who will hold shares $q \neq \varnothing$; and x_3 is the value to be shared, known to the input parties p. The input parties p and share parties q must all be present in the mode m, and no other parties may be present (so $m = p \cup q$). The resulting value is located at q, and has protocol $\texttt{enc}\#q$ indicating it is an encrypted value shared among parties q. The value to be shared may either be clear or encrypted among the parties p (i.e. $\vdash_p \psi$). When the value is already encrypted, the encrypted value is *reshared* from p to q without being decrypted.

A shared encrypted value is revealed from parties p to a nonempty set of parties $q \neq \varnothing$ as a cleartext result via the $\texttt{reveal}[x_1 \to x_2]\ x_3$, where x_1 evaluates to p, x_2 evaluates to q, and x_3 evaluates to the encrypted value, as described by rule ST-Reveal. All parties p among which x_3 is shared must be present, as well as the recipients of the value q, and no other parties.

The flexibility of rule ST-Share is a key strength of Symphony. By permitting $p \not\subseteq q$, we are able to support *clients*, which are parties in p but not in q. These clients *delegate* the secure computation to the parties q by providing input to the MPC but not participating thereafter. Clients and delegation are not directly supported by any existing MPC languages, nor is resharing. The `share` expressions in the `lwzShuffle` function of Figure 3.1 are (critically) both instances of resharing.

3.3 Distributed Semantics

This section presents $\lambda_{\mathbf{MPC}}$'s *distributed* (DS) semantics, modeling the communication and coordination needed for MPC. The next section proves the correspondence of the ST semantics w.r.t. the DS semantics.

3.3.1 Configurations

A *distributed configuration* C collects the execution states of the individual parties in an MPC. As shown at the top of Figure 3.6, it consists of a partial function from parties to *local configurations* $\dot{\varsigma}$, which are 5-tuples consisting of **(1)** a mode m, **(2)** a local environment $\dot{\gamma}$, **(3)** a local store $\dot{\delta}$, **(4)** a local stack $\dot{\kappa}$, and **(5)** an expression e. Local environments, stores, and stacks are the same as their ST counterparts except that instead of containing values v, they contain *local values* $\dot{v}$, which lack location annotations $@m$.

For a set of parties m wishing to jointly execute program e, the initial configuration C_0 will map each party $A \in m$ to a local configuration $(m, \varnothing, \varnothing, \top, e)$, where $\varnothing$ is the empty function (used for the empty environment and store), $\top$ is the empty stack, and e is the source program. Notice that each party tracks the *global* mode m in its local configuration.

3.3.2 Operational Semantics

The DS semantics $C \rightsquigarrow C'$ uses auxiliary judgments $\dot{\gamma} \vdash_m \dot{\delta}, a \hookrightarrow \dot{\delta}, \dot{v}$ and $\dot{\varsigma} \longrightarrow_A \dot{\varsigma}$; these are defined in part in Figure 3.6. The main rule DS-Step is used to execute a single party, independently of the rest. The rule selects some party A's local configuration $\dot{\varsigma}$, steps it to $\dot{\varsigma}'$, and then incorporates that back into the distributed configuration. This rule can be used whenever A's active expression e is anything other than `share` or `reveal`, which require synchronizing between multiple parties. Those two cases use the rules DS-Share and DS-Reveal, respectively, discussed below.

Non-synchronizing expressions The rules for relation $\dot{\gamma} \vdash_m \dot{\delta}, a \hookrightarrow \dot{\delta}, \dot{v}$ are essentially the same as those for the ST semantics, except that they operate on non-located data. The figure shows two examples. Rule ST-Var's conclusion locates the result at m via $\gamma(x)\downarrow_m$, but rule DS-Var's conclusion simply returns $\dot{\gamma}(x)$. Similarly,

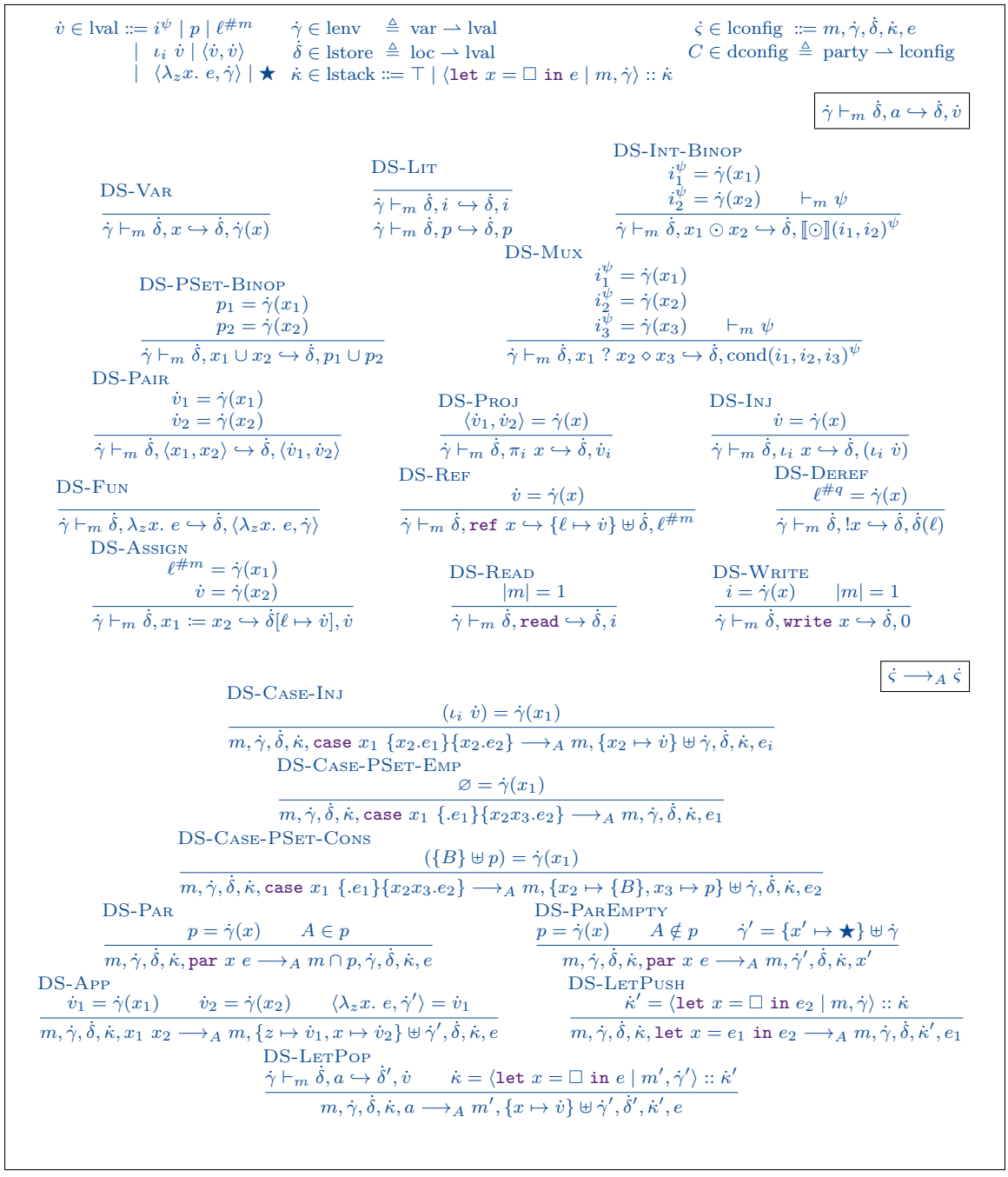


Figure 3.5: $\lambda_{\mathbf{MPC}}$ distributed semantics, local steps.

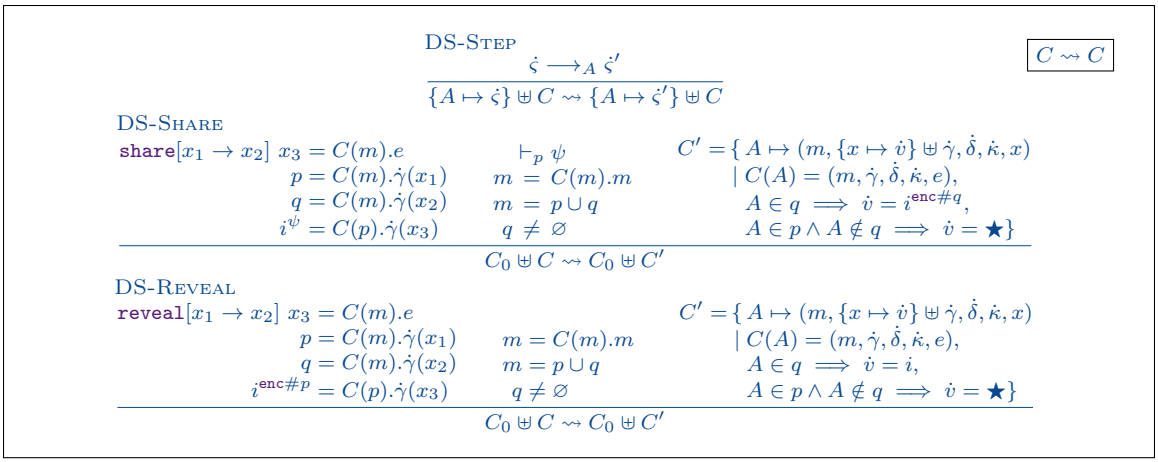


Figure 3.6: $\lambda_{\mathbf{MPC}}$ distributed semantics.

rule ST-IntBinop's premise requires $i_1^{\psi}@m = \gamma(x_1)\downarrow_m$ while rule DS-IntBinop's premise simply requires $i_1^{\psi} = \dot{\gamma}(x_1)$. For elimination forms, a location mismatch in a ST rule would translate to failed attempt to eliminate $\bigstar$ in the DS rule. For example, if rule ST-IntBinop would have failed because i_1^{ψ} was located not at m but at $p \subset m$ instead, then rule DS-IntBinop would fail for parties $A \in (m - p)$ since for these $\dot{\gamma}(x_1) = \bigstar$, which cannot be added to another share. The check $\vdash_m \psi$ is present in both rules to prevent attempts to add incompatible shares. Likewise, rules DS-Deref and DS-Assign (not shown) retain the check from the ST versions that the reference owners are compatible with m.

Judgment $\dot{\varsigma} \longrightarrow_A \dot{\varsigma}$ corresponds to ST judgment $\varsigma \longrightarrow \varsigma$, where annotation A indicates the executing local party. The rules for both judgments are essentially the same except for those handling $\mathtt{par}[x]\ e$. Rule DS-Par evaluates to the expression e so long as $A \in p$, where $p = \dot{\gamma}(x)$, updating the global mode to $m \cap p$, just as the ST semantics does. Rule DS-ParEmpty handles the case when $A \notin p$, thus skipping e and leaving global mode m as it is, evaluating to result x', which is a fresh variable bound to $\bigstar$ in $\dot{\gamma}'$.

Synchronizing expressions Rules DS-Share and DS-Reveal are used to evaluate expressions `share` and `reveal`, respectively. These expressions require synchronizing

between multiple parties, transferring data from one party to the other(s), so the rules manipulate multiple local configurations. But they are quite similar to their ST counterparts.

In the rules we write $C(m)$ to refer to the set of configurations mapped to by principals $A \in m$. When we write $C(m).e = e'$, we are saying that the expression component (e) of each configuration in the set $C(m)$ is equal to expression e'. For DS-Share, e' is $\texttt{share}[x_1 \to x_2]\ x_3$ and for DS-Reveal it is $\texttt{reveal}[x_1 \to x_2]\ x_3$. We similarly insist that each party's configuration agrees on the valuation of x_1 to p and x_2 to q, which together comprise the agreed-upon mode m. For DS-Share, the valuation of x_3 must be an integer with a protocol ψ compatible with p: $\vdash_p \psi$; for DS-Reveal, the valuation of x_3 for all sharing parties p must be an encrypted integer shared amongst those parties. These conditions are sufficient to guarantee that the *share* and *reveal* operations of the actual MPC backend complete successfully. [6] The updated configuration C' matches the original configuration C but updates the local configuration for each party $A \in m$ to have expression component x, where x is a fresh variable added to the store $\dot{\gamma}$ to map to the communicated (cleartext or encrypted) integer; those sharing parties $A \in p$ such that $A \notin q$ evaluate to $\bigstar$ instead.

3.4 Single-threaded Soundness

This section presents our main meta-theoretical results around *single-threaded soundness*, which is the sense in which we can interpret a $\lambda_{\mathbf{MPC}}$ program in terms of its ST semantics, even though in reality it will execute in a distributed fashion. Proofs are provided in Appendix A.2.1.

We relate a single-threaded configuration ς to a distributed one by *slicing* it, written $\varsigma\lightning$, which is defined in Figure 3.7. Each party A in the mode m of ς is mapped

[6] Note that in the actual implementation, each party $A \in m$ merely needs to check that its own view of m, p, and q is consistent per $m = p \cup q$—if not, as shown in the next section, it has landed in a *stuck configuration* and can signal that MPC has failed with a type error.

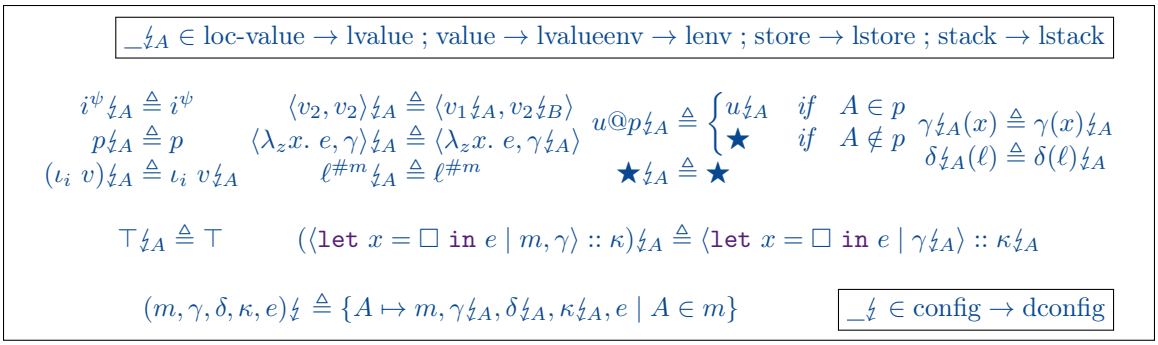


Figure 3.7: Slicing metafunction; relates ST and DS semantics.

to its local DS configuration consisting of m, expression e, and the sliced versions of the environment γ, store δ, and stack κ of ς that are specific to A. Slicing captures the simple idea that for a value $u@p$, if $A \in p$ then A can access u, but if $A \notin p$ then it cannot; $\_\!\downarrow_A$ works much like $\_\!\downarrow_{\{A\}}$ but strips off all location annotations.

We might hope to prove *full bisimulation* for the two semantics, but the backward direction (DS to ST) does not hold. Consider this program:

$$\texttt{let } x = \texttt{par}[A] \texttt{ <infinite loop> in par}[B]\ 1$$

In the DS semantics, party B's execution of the program can return 1 while A's loops. But such a distributed configuration cannot be "sliced to" from any ST execution, which always gets stuck in A's loop.

We can prove a full correspondence for programs whose execution trace concludes in normal form that is a *terminal state*.

Definition 3.4.1 (Terminal State)**.**

$$\begin{aligned}
\varsigma \text{ is a terminal state} &\iff \varsigma = m, \gamma, \delta, \top, a \wedge \gamma \vdash_m \delta, a \hookrightarrow \delta', v \\
\dot{\varsigma} \text{ is a terminal state} &\iff \dot{\varsigma} = m, \dot{\gamma}, \dot{\delta}, \top, a \wedge \dot{\gamma} \vdash_m \dot{\delta}, a \hookrightarrow \dot{\delta}', \dot{v} \\
C \text{ is a terminal state} &\iff \forall A \in dom(C).\ C(A) \text{ is a terminal state}
\end{aligned}$$

This definition captures the idea that a state is terminal if the execution stack is empty ($\top$), the next term to execute is atomic (a), and the atomic expression is able to step (via $\hookrightarrow$) to a value v. There are no successor configurations which can be reached from a terminal state. Any state which is both non-terminal and also has no

successor configurations we call *stuck*.

Theorem 3.4.1 (ST/DS Terminal Correspondence)**.** *If* $\varsigma \longrightarrow^* \varsigma'$*, then the following statements imply one another for any* C*:*

1. ς' *is a terminal state and* $C = \varsigma' \lightning$

2. $\varsigma \lightning \rightsquigarrow^* C$ *and* C *is a terminal state*

The proof follows from a *forward simulation* lemma, which establishes that for every single-threaded execution there exists a compatible distributed one, and *confluence*, which establishes that even though the distributed semantics is nondeterministic, its final states are uniquely determined.

What about executions which diverge or get stuck? We prove two useful theorems about these.

Theorem 3.4.2 (ST/DS Strong Asymmetric Non-terminal Correspondence)**.** *The following statements are true:*

1. *If* $\varsigma \lightning$ *reaches a stuck state (under* $\rightsquigarrow$*) then* ς *reaches a stuck state (under* $\longrightarrow$*)*

2. *If* ς *divergent (under* $\longrightarrow$*) then* $\varsigma \lightning$ *divergent (under* $\rightsquigarrow$*)*

Theorem 3.4.2 does not rule out the possibility that ς gets stuck while $\varsigma \lightning$ never does. Consider this example

$$\texttt{let } x = \texttt{par}[A] \texttt{ <error> in par}[B] \texttt{ <infinite loop>}$$

In the ST semantics, this program gets stuck. In the DS semantics, A will only become *locally stuck* while B runs forever. We can prove that if a single-threaded configuration gets stuck, then for any reachable distributed configuration there exists a reachable, locally stuck state:

Theorem 3.4.3 (ST/DS Soundness for Stuck States)**.** *If* $\varsigma \longrightarrow^* \varsigma'$ *and* ς' *is stuck, then for every* C *where* $\varsigma\lightning \rightsquigarrow^* C$ *there exists a* C' *s.t.* $C \rightsquigarrow^* C'$ *and* C' *is locally stuck.*

It follows that if the ST semantics applied to ς detects a runtime error (i.e., gets stuck), then (assuming a non-pathological scheduler) one of the local configurations of $\varsigma\lightning$ will eventually detect a runtime failure (i.e., get locally stuck), too.

In sum: the ST and DS semantics correspond for terminating programs; they correspond for non-terminating programs too, but with a local notion of "stuckness" applied to DS states.

3.5 Implementation

We implemented a Symphony interpreter in 4K lines of Haskell. The interpreter can run programs in sequential mode for prototyping and debugging, and distributed mode for actual MPC. Symphony adds a number of features beyond $\lambda_{\mathbf{MPC}}$, including booleans and a conditional expression; nested pattern-matching on pairs, sums, lists, principal sets and bundles; arrays (mutable vectors with O(1) lookup); synchronized randomness; and implicit embedding of constants as shares.

We have implemented a standard library (about 800 LOC) for Symphony that includes various data structures and coordination patterns, e.g., initializing a bundle from a principal set, and bounded recursion for unrolling an MPC function.

In addition to the Symphony interpreter, we have also implemented a Symphony runtime system in 2K lines of Rust. The runtime system acts as a compatibility layer between the interpreter and MPC backends. The runtime provides a common interface to the interpreter by enhancing MPC backends with missing features that Symphony requires. The runtime supports MPC based on Yao's garbled circuit protocol using EMP toolkit [176], and the N-party GMW protocol using MOTION [36].

Both Yao's protocol and GMW are semi-honest and evaluate boolean circuits.

3.5.1 Interpreter

As briefly mentioned, the SYMPHONY interpreter adds a number of interesting features beyond $\lambda_{\mathbf{MPC}}$. In this subsection, we discuss three of the most interesting and important features: synchronized randomness, MPC over algebraic data types, and the standard library.

Note that the `share` primitive is extended in SYMPHONY as: `share`$[\phi, \tau : P \to Q]\ v$. Here, the metavariable ϕ denotes protocols and the metavariable τ denotes types. A particular protocol implementation must provide operations for encrypting, computing over, and decrypting base values.

Synchronized Randomness Symphony provides a convenient way to generate randomness across multiple parties in parallel, ensuring that all parties receive the same random value. This functionality can be implemented as a library using only access to local randomness, as shown in the $\lambda_{\mathbf{MPC}}$ code below.

```
1  def randomSend P n =
2    let rec sum' = fun Q ->
3      case Q
4      { {}          -> 0n
5      ; { p } \/ Q' ->
6        send [nat : p -> P]
7          (par { p } rand { p } nat) + (sum' Q')
8      }
9    in
10   let sum = sum' P in
11   sum % n
```

This code will generate a random value on each party in `P` and sum them all together, modulo `n`, to generate a random natural number in the range $0..n-1$. This works, but it is inefficient because it requires communication between all the parties each time they wish to generate a synchronized random number.

The primitive expression provided by SYMPHONY, `rand P` μ (where μ is a base type like `nat`), provides the same functionality but does so more efficiently. This is implemented using a shared, cryptographic PRNG among the parties P.

MPC over Algebraic Types The SYMPHONY interpreter also generalizes $\lambda_{\mathbf{MPC}}$ by allowing `mux`, `case`, `share` and `reveal` to operate recursively over on pairs, sums and lists. It adds `mux-case` for case analysis on encrypted sums, which are represented with pairs: $\lambda_{\mathbf{MPC}}$ value $\iota_0\ v$ is represented as $\langle \texttt{true}, \langle v, \texttt{default}\rangle\rangle$ and value $\iota_1\ v$ is represented as $\langle \texttt{false}, \langle \texttt{default}, v\rangle\rangle$, with each of the components encrypted. The value `default` is to allow case analysis to proceed on *both* branches of `mux-case`, as a kind of multiplexor. The precise value of `default` is determined when sharing, based on the type annotation τ on `share`$[\phi,\tau{:}P{\to}Q]$ `default`.

SYMPHONY generalizes $\lambda_{\mathbf{MPC}}$'s `share`, `mux`, `case`, and `reveal` by allowing arbitrary algebraic types as arguments. It also adds another expression, `mux-case`, for case analysis on encrypted (shared) values. The `share`, `mux`, `case`, and `reveal` expressions on pairs are generally unsurprising. Sharing a pair is implemented by recursively sharing, component-wise.

$$\texttt{share}[\phi, \tau_1 \times \tau_2 : P \to Q]\ (a,\ b) = (\texttt{share}[\phi, \tau_1 : P \to Q]\ a,\ \texttt{share}[\phi, \tau_2 : P \to Q]\ b)$$

The `mux`, `case`, and `reveal` operations on pairs are similar.

Sums are represented as tagged pairs. The $\lambda_{\mathbf{MPC}}$ value $\iota_0\ v$ is represented as $\texttt{sum}\langle \texttt{true}, v, \texttt{default}\rangle$ and the value $\iota_1\ v$ is represented as $\texttt{sum}\langle \texttt{false}, \texttt{default}, v\rangle$. The value `default` is a placeholder which will be replaced by a value of the appropriate type if the sum value is shared. Sharing a sum value is implemented, as with pairs,

by recursively sharing, component-wise.

$$\texttt{share}[\phi, \tau_1 + \tau_2 : P \to Q]\ \texttt{sum}\langle b, v_1, v_2\rangle = \texttt{sum}\langle \texttt{share}[\phi, \mathbb{B} : P \to Q]\ b,$$
$$\texttt{share}[\phi, \tau_1 : P \to Q]\ v_1,\ \texttt{share}[\phi, \tau_2 : P \to Q]\ v_2\rangle$$

To share a `default` value, $\texttt{share}[\phi, \tau : P \to Q]\ \texttt{default}$, we simply share instead an arbitrary default value of the appropriate type τ. In practice, we choose the identity value (e.g., `false` for $\mathbb{B}$, 0 for $\mathbb{Z}$) but, as we will see shortly, the choice doesn't matter because it will never be observable. The `mux` and `reveal` operations on sum values work similarly.

The $\texttt{case}\ v\ \{\theta_1 \to e_1; \ldots; \theta_n \to e_n\}$ expression works mostly as expected, iterating through the list of patterns, $\theta_1, \ldots, \theta_n$, and checking if there is a `clear-match` of v against the current pattern θ_i. When the first match is encountered, the environment is suitably extended[7] and execution proceeds with the corresponding body e_i. The only caveat to this standard description of pattern matching is that an *encrypted* sum value does not `clear-match` left or right injection patterns. Since it is encrypted, we cannot inspect the tag. This observation motivates SYMPHONY's `mux-case` expression which does permit matching on encrypted sum values.

The $\texttt{mux-case}\ v\ \{\theta_1 \to e_1; \ldots; \theta_n \to e_n\}$ expression works by mapping the following procedure over the list of patterns, $\theta_1, \ldots, \theta_n$. First, check if there is an `enc-match` of v against the current pattern θ_i. The `enc-match` check is exactly like `clear-match` except that any sum value matches both the left injection and right injection patterns. Then, evaluate the corresponding body e_i to v_i in the environment suitably extended. If the pattern is **not** a left or right injection, then return v_i. If the pattern is is a left injection, then return $\texttt{mux}\ b\ \texttt{then}\ v_i\ \texttt{else}\ \texttt{default}$. If the pattern is a right injection, then return $\texttt{mux}\ b\ \texttt{then}\ \texttt{default}\ \texttt{else}\ v_i$. We now have a list of values, $v_1, \ldots, v_m$ ($m \leq n$), which we add together according to an `add` procedure which is defined

[7] For example, a `clear-match` of $\texttt{sum}\langle\texttt{true}, v_l, v_r\rangle$ against the pattern $\iota_{\texttt{true}}\ x$ would extend the environment with $[x \mapsto v_l]$

for values which have the same shape. The `add` procedure simply combines values recursively, using an appropriate additive operation for each of the base values (e.g., $\texttt{add}\ (\texttt{false}, 0)\ (\texttt{true}, 42) \equiv (\texttt{add}\ \texttt{false}\ \texttt{true}, \texttt{add}\ 0\ 42) \equiv (\texttt{false} \vee \texttt{true}, 0 + 42) \equiv (\texttt{true}, 42)$). The `default` value is an identity for `add`.

Consider the standard encoding of booleans as a sum of units: $\texttt{bool} \coloneqq \texttt{unit} + \texttt{unit}$. We will take $\texttt{sum}\langle\texttt{true}, \bullet, \bullet\rangle$ to be the encoding of `true` and symmetrically for `false`. We would hope that the obvious encoding of `mux` in terms of `mux-case` would work: $\texttt{mux}\ \texttt{sum}\langle b, \bullet, \bullet\rangle\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \coloneqq \texttt{mux-case}\ \texttt{sum}\langle b, \bullet, \bullet\rangle\ \{\iota_{\texttt{true}}\ \bullet \to e_1;\ \iota_{\texttt{false}}\ \bullet \to e_2\}$. Indeed, it does:

$$
\begin{aligned}
&\texttt{mux-case}\ \texttt{sum}\langle\texttt{true}, \bullet, \bullet\rangle\ \{\iota_{\texttt{true}}\ \bullet \to e_1;\ \iota_{\texttt{false}}\ \bullet \to e_2\} \\
&\equiv \texttt{add}\ (\texttt{mux}\ \texttt{true}\ \texttt{then}\ e_1\ \texttt{else}\ \texttt{default}) \\
&\qquad\quad (\texttt{mux}\ \texttt{true}\ \texttt{then}\ \texttt{default}\ \texttt{else}\ e_2) \\
&\equiv \texttt{add}\ e_1\ \texttt{default} \\
&\equiv e_1
\end{aligned}
$$

The Standard Library SYMPHONY allows programmers to write complex distributed programs in a natural way. The standard library shipped with SYMPHONY has many of the usual fixings of functional programming languages such as libraries for options; eliminators (folds) over various algebraic types (nats, options, lists, maps, etc.); higher order functions (flip, compose, curry, uncurry, etc.). However, it also has some unusual fixings, such as libraries for coordination and secure recursion.

The coordination library primarily addresses common operations on principal sets, bundles, and their interaction. For example, Figure 3.8 contains a function, `mapPairs`, which takes a function `f`, principal set `P`, and bundle `b`, and non-deterministically pairs up principals in `P` before executing `f` with their respective inputs from `b`, returning all the results as a list. This function could be used to pair up parties who then compete

in a 2-player game using MPC, in a tournament-style application.

Another function in Figure 3.8, `solo-f`, takes a principal set `P` and a function `f` and runs `f` at every principal ρ in `P`. This function is used to define the `bundleInputs` function.

```
def bundleInputs P = solo-f P (fun _ -> read nat)
```

The last function in Figure 3.8, `unroll`, takes an "almost" recursive function and unrolls it `n` times. If the function would recur more than `n-1` times, it uses `init` as the value for the recursive call at recurrence `n`. The "almost" recursive function is defined using `brec` and is just like a recursive function, except that recursive calls are replaced with calls to a higher-order argument. The expectation is that the caller of this function will provide the function itself as the higher-order argument to perform a finite unrolling of the function. For example, here's a recursive GCD function:

```
def gcd (a, b) = if (a == 0) then b else gcd (b % a, a)
```

and here's its "almost" recursive counterpart:

```
def gcd-almost f (a, b) = if (a == 0) then b else f (b % a, a)
```

The `unroll` function is useful for defining a finite unrolling of a recursive function so it may be computed in MPC.

```
def gcd-mpc f (a, b) = mux if (a == 0) then b else f (b % a, a)
```

As mentioned earlier, SYMPHONY provides syntactic sugar, via the `brec` keyword, which makes this look even more natural:

```
def brec gcd-mpc (a, b) = mux if (a == 0) then b else gcd-mpc (b % a, a)
```

If we now want to compute GCD to a maximum of 32 iterations, we can do so.

```
def gcd-mpc-32 = unroll gcd-mpc (const 0) 32n
```

```
def mapPairs P b f = case P
  { {}                  -> []
  ; {p1} \/ {p2} \/ P' ->
    let r = f p1 b@p1 p2 b@p2 in
    r :: mapPairs P' b f
  }

def solo-f P f = case P
  { {}        -> <<>>
  ; {p} \/ P' -> << p | par {p} f p >> ++ solo-f P' f
  }

def unroll f init n =
  if n == 0n then init
  else f (unroll f init (n - 1n))
```

Figure 3.8: Selected functions from the standard library of SYMPHONY.

3.5.2 Runtime System

As briefly mentioned, the SYMPHONY runtime system acts as a compatibility layer between the SYMPHONY interpreter and MPC backends. In this subsection, we discuss how the runtime system adds support for delegation, resharing, and reactive MPC. A detailed description of the implementation that addresses engineering details such as the FFI and resource management can be found in Appendix A.1.

Delegation and Resharing The runtime adds support for delegation and resharing via semi-honest XOR sharing over SYMPHONY parties. For example, A delegates to B, C by generating two XOR shares of her input and sending those shares to B and C respectively. Then, B and C convert their shares into the native representation of the backend by encrypting them and combining the encrypted shares into an encrypted value using XOR. At this point, A's original input is natively encrypted among B and C. Similarly, B, C *reshare* to D, E, F by converting the encrypted value into XOR shares and decrypting them to B and C respectively. Then, B and C delegate their shares to D, E, F which D, E, F combine using XOR. At this point,

the value originally natively encrypted among B and C is natively encrypted among D, E, and F.

The benefit of these procedures is that they are *generic*, treating the underlying MPC backend as a black box by relying only on standard features. Of course, the drawback of these procedures is also that they are *generic*, failing to take advantage of optimizations available for specific protocols.

The SYMPHONY runtime does not rely on generic conversion when converting shares to and from MOTION's native representation, since MOTION uses shares natively. The SYMPHONY runtime does rely on generic conversion when converting shares to EMP's native representation, but does use the optimal conversion (`Y2B` [54]) when converting from EMP back to shares. The optimal conversion from EMP to shares is supported by EMP, but the optimal conversion from shares to EMP (`B2Y` [54]) is not. Efficient conversion procedures for 2-party and N-party MPC protocols have been identified between Boolean sharing, Arithmetic sharing, and Boolean garbling [54, 36].

Reactive MPC The runtime also adds support for reactive MPC to MOTION. An MPC context in MOTION, called a `Party`, is a C++ object that manages the global state associated with the MPC. It contains information about the current party that is executing, the total number of parties, how parties can be contacted (e.g. via TCP sockets), a shared PRG, and a binary circuit composed of gates and wires. When an encrypted value is created or computed, the `Party` object creates a new gate which is added to the circuit. The actual encrypted value is a reference to the output wire of the new gate. When the programmer has finished their MPC, they instruct the `Party` object to execute the underlying circuit. At that point, the circuit is executed using GMW and an XOR share can be extracted from any encrypted value.

Unfortunately, after the `Party` is executed, we are no longer allowed to use this

`Party` object for additional MPC: the `Party` object is effectively defunct. The ability to continue performing MPC is precisely what is needed to support reactive MPC, in which values are decrypted and then used to influence additional computation.

The runtime adds support for reactive MPC to MOTION by caching the XOR shares resulting from one execution of a `Party` object, destroying it, and then creating a new one. When we compute on encrypted values produced by the previous `Party`, we provide them as input to the new `Party` before proceeding with the computation. While conceptually simple, it required considerable engineering to achieve acceptable performance. For example, we had to modify MOTION to add support for creating a `Party` object using existing TCP connections (rather than MOTION creating its own). This ensured that TCP connections were only established once for each pair of SYMPHONY parties, rather than repeatedly by MOTION whenever a new `Party` object was created (i.e. on each decryption).

3.6 Experimental Evaluation

This section shows, through a series of experiments and case studies, that SYMPHONY provides superior programming expressiveness and ergonomics compared to prior systems, while maintaining competitive performance.

3.6.1 Expressiveness and Ergonomics

We discuss SYMPHONY's expressiveness and ergnomics benefits based on our experience implementing sixteen programs from the MPC literature. The programs are tabulated in Table 3.1. As points of comparison we consider if (or whether) versions of these programs could be implemented in Wysteria [143, 144] and/or Obliv-C [181], a state-of-the-art two-party MPC framework for C, and whether they are N-party or 2-party programs. We categorize the SYMPHONY language features required to

Table 3.1: A collection of implemented MPC programs. # indicates the number of parties. **Lang.** indicates the implementation language: SYMPHONY (`S`), Obliv-C (`O`), or Wysteria (`W`), where `?` means the program *should* be supported but we have no example, and $*$ means a 2-party (rather than N-party) version is supported. **Features** indicates the features required to implement the program: `P` for `par`, `R` for reactive MPC, `$` for synchronized randomness, `D` for delegation, and `S` for resharing. **L (C)** indicates the lines of code (resp. characters) of the SYMPHONY program measured using `wc -l` (resp. `wc -m`).

Program	#	Lang.	Features	L (C)	Description
`hamming` [**10.1007/978-3-642-54807-9_15**, 188]	2	`S,O,W?`		19 (629)	Find the Hamming distance of two strings.
`edit` [181]	2	`S,O,W?`		56 (1574)	Find the edit distance, by dynamic programming, of two strings.
`bio-match` [39]	2	`S,O,W?`		38 (949)	Compute the minimum Euclidean distance between a set of points (from A) and a single point (from B) in 2D space.
`db-analytics` [39]	2	`S,O,W?`		121 (3020)	Compute the mean and variance over the union and join of two databases.
`gcd` [**10.1145/3319535.3339818**]	2	`S,O,W`		13 (416)	Compute the GCD of two numbers via Euclid's algorithm.
`richest` [143]	N	`S,W`		8 (176)	An N-party variant of the Millionaire's Problem.
`gps` [143]	N	`S,W`		40 (1213)	Compute the one-dimensional nearest neighbor for each of N parties.
`auction` [143]	N	`S,W`		34 (920)	Compute a second-price auction, revealing the second-highest bid to everyone and the highest bidder to auctioneer.
`median` [143]	2	`S,O?,W`	`R`	26 (566)	Compute the mixed-mode (reactive) median of a set of numbers.
`intersect` [143]	2	`S,O?,W`	`R`	17 (563)	Compute a naive private set intersection over two sets.
`tournament`	N	`S`	`P,$,R`	34 (760)	Use a comparison-based single-elimination tournament to find the richest of N parties.
`committee`	N	`S`	`P,D,$`	45 (946)	Elect a small committee of size $K < N$, which is useful for fair delegation MPC from N to K parties.
`waksman` [173, 60]	N	`S,O*`	`$`	209 (5947)	Securely shuffle of an array, using N iterations of a Waksman permutation network.
`lwz` [105] (Section 3.1.2)	N	`S`	`P,$,S`	23 (745)	Securely shuffle of an array, using N reshares of a linear secret sharing scheme (LSSS).
`trivial-doram` [71, 38]	N	`S,O*,W?`		55 (1883)	A library for Oblivious RAM, adapted to MPC from trivial client-server ORAM.
`shuffle-qs` [79]	N	`S`	`P,D,$,S,R`	62 (1742)	Securely sort using Shuffle-Then-Sort with `lwz` as the underlying secure shuffle and QuickSort as the sorting algorithm.

express the programs in the **Features** column:

- *Coordination* (`P`). Coordination of parties that would be difficult and error-prone in languages without explicit support for `par` expressions.
- *Reactive MPC* (`R`). Secure computation that depends on the decrypted result of a previous secure computation.
- *Synchronized Randomness* (`$`). Cryptographically secure random number generation which is synchronized among a set of parties (i.e. all parties agree on the number generated).
- *Delegation* (`D`). A party encrypts a value to, or decrypts a value from, a secure computation in which she does not participate.
- *Resharing* (`S`). A value encrypted among a set of parties is transferred to a different set of parties without being decrypted.

`tournament` orchestrates a single-elimination tournament over N parties. Each match in the tournament is a 2-player secure computation, with the winner moving on to the next round. The program requires *Coordination* because the participants in each match are dynamically assigned from the set of remaining players. Likewise, the set of remaining players is determined dynamically according to outcome of the matches in the previous round. It requires *Synchronized Randomness* to determine the participants in each match. Finally, it requires *Reactive MPC* to decrypt the winners in each round so that they may be coordinated in the following round.

`committee` performs an arbitrary secure computation among N parties by selecting a random committee of size $K < N$ and delegating the secure computation to them. The program requires *Coordination* and *Synchronized Randomness* because the committee is computed dynamically according to K and synchronized random

numbers generated by the N parties. It requires *Delegation* because the parties outside the committee encrypt their input and send it to the committee but do not participate in the secure computation.

`waksman` and `lwz` are protocols for securely shuffling an array of elements among N parties without revealing the permutation to any of them. We discuss these protocols in much more depth and compare their performance in Section 3.6.2. Both programs require *Synchronized Randomness* to perform (non-secure) shuffles as a subroutine. `lwz` additionally requires *Coordination* and *Resharing* to compute all the party sets of size $N - T$ and reshare the array of elements to and from these subsets.

`shuffle-qs` is the most sophisticated program, using both `committee` and `lwz` as subroutines. It implements a secure sorting procedure over N parties by choosing a committee of size $K > 2$, shuffling the elements among the committee using `lwz`, and then sorting the elements with QuickSort by revealing the result of each comparison. It requires *Coordination*, *Delegation*, *Synchronized Randomness*, and *Resharing* by virtue of relying on `committee` and `lwz`. It requires *Reactive MPC* because each comparison in QuickSort is decrypted before securely computing the next comparison.

Expressiveness Of the sixteen programs, we believe that Obliv-C can express 9 and Wysteria can express 11. This is because both languages lack some needed features.

Obliv-C only supports two parties, ruling out fundamentally N-party programs. Obliv-C also lacks clean support for *Coordination*, as discussed in Section 3.1.

Wysteria does not have support for *Synchronized Randomness*, *Delegation*, or *Resharing*. These limitations make it impossible to express `tournament`, `committee`, `waksman`, `lwz`, and `shuffle-qs`. Extending Wysteria with *Synchronized Randomness* would be straightforward, which would allow Wysteria to express `waksman`. The other programs still have barriers: `lwz` and `shuffle-qs` require *Delegation* or *Resharing*, and `tournament` requires sophisticated coordination mechanisms that Wysteria's

Figure 3.9: The `gcd` program of Table 3.1 as written in SYMPHONY (left) and Wysteria (right).

```
1   def brec gcdr a b =
2     mux if (a == 0) then b
3     else gcdr (b % a) a
4   def gcd = unroll gcdr (const 0) 93
5
```

```
1   let gcdr i sa sb =
2     if i == 0 then
3       let ret =sec({A,B})=
4         combsh sb in ret
5     else
6       let (sa', sb') =sec({A,B})=
7         let (a, b) =
8           (combsh sa, combsh sb) in
9         if (a == 0) then
10          (makesh a, makesh b)
11        else
12          (makesh (b % a), makesh a)
13      in gcdr (i - 1) sa' sb'
14  let gcd = gcdr 93
15
```

static type system does not accept. It uses on refinement types over subset constraints to ensure coordination is performed safely, but its decisions can be over-conservative. The first and third authors spent a few hours each trying to write the `subsets` function of `lwz` (Section 3.1.2, Figure 3.1) in Wysteria, which is representative of computations in `tournament`, but were not able to get it past the typechecker.

Ergnomics SYMPHONY can also provide ergonomics benefits even when a program could be expressed in another language. This is especially true when considering Wysteria, which also has coordination features.

To illustrate, consider `gcd`. This program computes the GCD of two encrypted values among `{A,B}`. It nicely leverages SYMPHONY's first-class shares to allow the code to be straightforward. By contrast, Wysteria relies on special `sec` blocks for secure computation, which can make programming awkward. Figure 3.9 shows GCD in the two languages side by side. Conceptually, the entire `gcd` function is executed under secure computation. Wysteria does not allow function calls within a `sec` scope. As a result, we are forced to enter and exit `sec` mode (line 6) on each recursive call to `gcdr`. In addition to being verbose, this pattern can also have performance

implications. Each time execution enters a `sec` block, the expression (lines 7-12) is compiled and executed as a circuit. The overhead of compilation and circuit execution is repeated on each iteration. In contrast, the SYMPHONY version of `gcd` constructs the entire circuit for GCD before execution. While the asymptotic performance is identical, the measured performance is likely to be worse in Wysteria due to the constant overhead mentioned above. Unfortunately, we were unable to empirically validate this claim. We have only been able to get Wysteria to typecheck the programs above, not actually run them.

As mentioned above, the conservativeness of Wysteria's type system can also make programming awkward, even if it does not ultimately prevent expressing a program entirely.

3.6.2 Performance

SYMPHONY's expressiveness allows programmers to write optimized protocols directly and simply. For example, the `median` and `shuffle-qs` programs leverage *Reactive MPC* to perform certain comparison operations in the clear, dramatically improving performance over a monolithic protocol [93, 145]. While Reactive MPC is available in some languages, the combination of features needed to express the `lwz` secure shuffle is unique to SYMPHONY—no other language can express it. Compared to `waksman`, another secure shuffle algorithm, `lwz` offers a substantial performance benefit because it requires *no computation under cryptography*, just re-sharing and comparisons/shuffling in the clear. As shown in Section 3.6.2, the result is dramatically faster running times for all input sizes.

SYMPHONY's expressiveness does not place an undue burden on the implementation's ability to achieve good performance. We compared SYMPHONY's performance with that of Obliv-C, a highly optimized MPC framework. As described in detail in Section 3.6.2, we ran on the first five programs in Table 3.1, which are well known and

frequently referenced in the literature. We configured both Symphony and Obliv-C to use EMP's [176] 2-party garbled circuits implementation, to isolate language overhead from cryptography costs. On a simulated LAN under MPC, Symphony's running time was 1.15× that of Obliv-C. Without MPC, Symphony time is 2.4× that of Obliv-C. On a WAN (limited to 100 gbps and 50 ms RTTs), Symphony time was 0.85× that of Obliv-C. Examining these overheads, we find that one source is Symphony's support for $N > 2$ parties: party inclusion checks require the use of sets (implemented as balanced binary trees) rather than simple equality tests. The more significant overhead is unrelated to Symphony itself: the language is implemented as an interpreter in Haskell, whereas Obliv-C is embedded within compiled C. Indeed, looking at the sizes of garbled circuits generated by both frameworks, we found them to be very similar. Thus, we would expect even closer performance in a more production-quality implementation.

Utility: Waksman vs. LWZ in Symphony

We identified the LWZ shuffle (Section 3.1.2, Figure 3.1, `lwz` in Table 3.1) as one of the programs that can only be expressed in Symphony. In this section we demonstrate the utility of the LWZ shuffle by comparing it to the Waksman shuffle (`waksman` in Table 3.1), which is the de-facto protocol for performing a secure shuffle in most MPC languages. We show that the empirical performance of Waksman and LWZ in Symphony matches the expected asymptotics. Based on these results, we conclude that the added expressiveness of Symphony manifests not only as convenience and safety, but can also manifest as *performance* when the optimal protocol (e.g. LWZ) for a given task is difficult or impossible to express in another language.

Waksman The Waksman protocol implements a secure shuffle via repeated application of a classic Waksman *permutation network* [173]. A permutation network

repeatedly and conditionally swaps two list elements at a time until the list is fully shuffled; each swap can be implemented from Boolean gates. One subtlety of a permutation network is that one must choose the *control bits* of the network, which is an input that dictates which of the swap gates should indeed swap their input. To fully hide the shuffle from all parties, each party secretly chooses their own control bits and programs one of the sequence of $|P|$ networks. Thus, the full shuffle requires $|P|$ networks, and the implementation requires some coordination: the programmer must prescribe that each party will, one-by-one, program a network.

Security vs. Performance The Waksman and LWZ protocols present different tradeoffs in terms of security and performance.

Given an input list of n integers of bitwidth w, a Waksman permutation network is a recursive algorithm requiring $O(w \cdot n \log n)$ Boolean gates. Since we must repeat the network $|P|$ times, we require $O(|P| \cdot w \cdot n \log n)$ gates total. The network's circuit depth grows with $O(\log n)$, which is relevant since the round complexity of interactive MPC protocols, such as GMW, grows with depth; in total we need $O(|P| \cdot \log n)$ rounds of communication.

The LWZ protocol, on the other hand, avoids the need for general purpose MPC circuit evaluation. Indeed, the protocol is strikingly lightweight: The LWZ protocol does not require execution of any secure gates at all. The downside of LWZ is that its performance degrades with the number of *tolerated corruptions*, t. I.e., suppose that at most t parties will collude and share information with one another. To prevent these adversaries from learning the final permutation of the elements, we must ensure that for *each* subset of t parties, there exists one repetition of the protocol where *none* of those parties is on the committee. Thus, we must make our committees each of size $|P| - t$, and the number of needed repetitions grows with $\binom{|P|}{|P|-t}$. If t is small, say $t = 1$, then the LWZ protocol has excellent performance requiring only $|P|$ rounds of

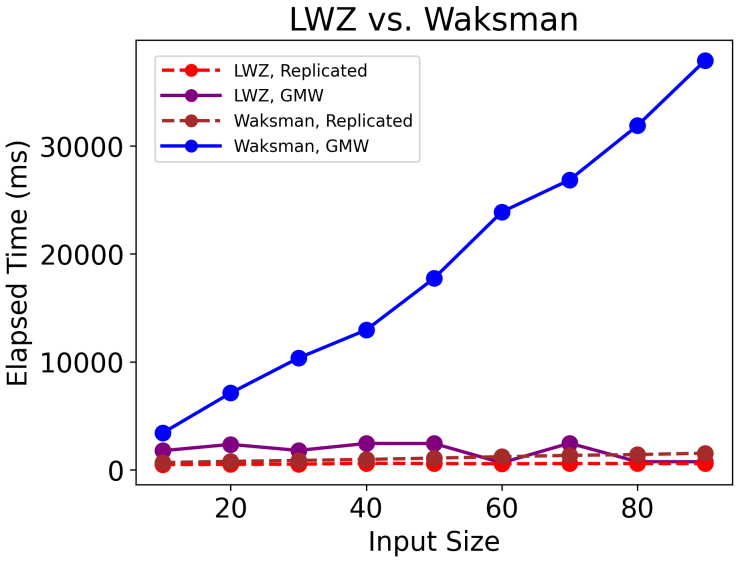


Figure 3.10: End-to-end execution time of Waksman vs LWZ shuffle over three parties, averaged over five samples (lower is better). The **Replicated** protocol executes the program without cryptography, executing operations in the clear. The **GMW** protocol uses the SYMPHONY implementation of GMW which uses MOTION as a backend. **Input Size** indicates the length of the integer list provided as input by each party.

communication. If t is large, say $t = \frac{|P|}{2} - 1$, then performance degrades exponentially in $|P|$.

SYMPHONY Execution Time Figure 3.10 plots SYMPHONY's end-to-end execution time for the LWZ and Waksman shuffles. In both protocols, three parties each share an array of **Input Size** integers which are concatenated and shuffled. We ran the programs using both the **GMW** protocol using the **Replicated** protocol as a baseline. In the **Replicated** protocol, Boolean gates are implemented locally and computed in the clear instead of via cryptography. For our LWZ threshold, we chose the optimistic setting where the maximum number of colluding parties is $t = 1$.

Our results demonstrate that the SYMPHONY implementation of LWZ properly avoids MPC overhead: as already stated, LWZ is a lightweight protocol, so SYMPHONY should not – and does not – erroneously introduce cost just because we are operating on GMW shares. As expected, we find that the **GMW**-based Waksman

implementation is much slower than both **Replicated** Waksman and both variants of LWZ. The slowdown is primarily due to the cryptography required to execute the Boolean gates under MPC.

We do note that **GMW**-based Waksman achieves lower performance than might be expected. We observed that the low performance is due to the MOTION backend which, on this benchmark, allocates $>$ 4GB of memory per party to store the GMW circuit. Moreover, the execution of each gate involves accessing many non-contiguous memory addresses, leading to low spacial locality. We believe that performance can be greatly increased by handling more of the circuit generation and execution in the compatibility layer of SYMPHONY.

Even with a highly optimized GMW backend, the LWZ protocol would remain best for the setting of $t = 1$: the coordination-heavy LWZ protocol is simply a superior technique for the setting. SYMPHONY's features make the complex coordination involved in this protocol easy to express.

SYMPHONY vs. Obliv-C

We compared SYMPHONY's performance, in terms of running time and gate counts, against that of Obliv-C on the same programs.

Experimental Setup For fair comparison, both SYMPHONY and Obliv-C were configured to use EMP [176] as their MPC backend; we extended Obliv-C to use EMP via its callback interface. We used both SYMPHONY and Obliv-C to implement a benchmark suite of five programs: `hamming`, `edit-dist`, `bio-match`, `db-analytics`, and `gcd`. See Table 3.1 for a description of these programs.

Experiments were run on a 2019 MacBook Pro with a 2.8 GHz Quad-Core Intel Core i7 and 16GB of RAM (OSX 11.3.1). The Obliv-C compiler is an extension of GCC 5.5.0, and all benchmarks were compiled with `-O3` optimizations. Experiments

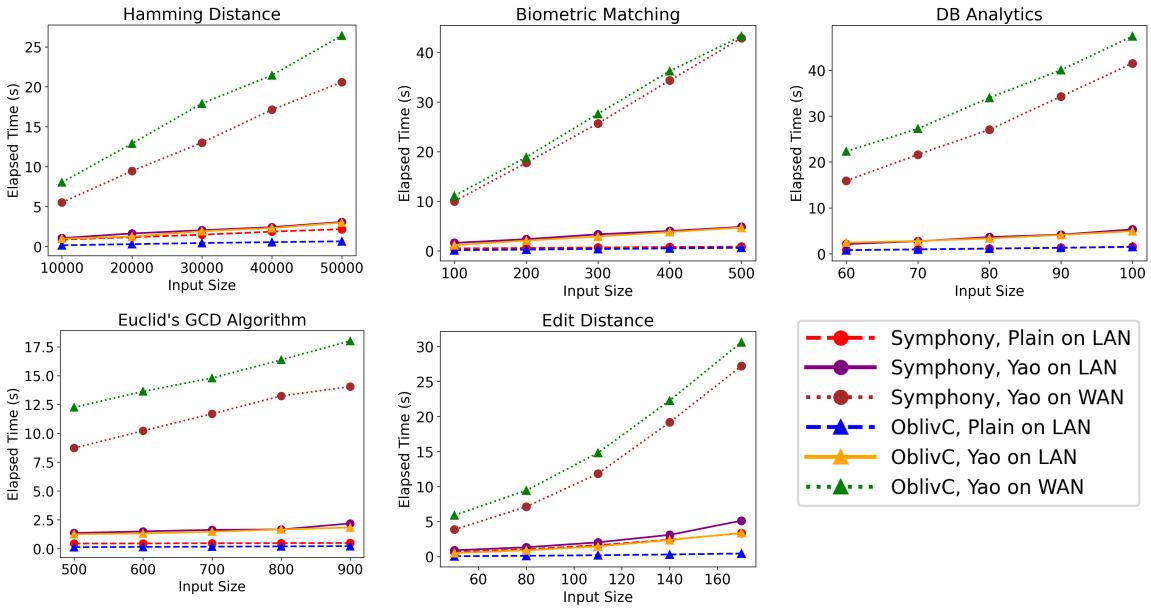


Figure 3.11: End-to-end execution time of 5 programs, averaged over five samples (lower is better). **LAN** is a simulated 1gbps connection with no delay. **WAN** is a simulated 100Mbps connection with a 50ms RTT latency. **Yao** and **Plain** protocols use EMP's `sh2pc` (semi-honest, two-party) and `plain` protocols respectively. SYMPHONY uses EMP's Integer interface. Obliv-C uses EMP's Bit interface (compiles integer operations to circuits). Input sizes for all the benchmarks indicate the length of the list(s) provided as input, except for `gcd-gc` where the input size indicates the number of iterations of the GCD algorithm.

were run on two simulated networks: a LAN (1Gbps bandwidth, <1ms RTT latency) and a WAN (100Mbps bandwidth, 50ms RTT latency). All experiments use 32-bit integers, except for `gcd-gc` which uses 64-bit integers. Reported execution times measure the end-to-end execution time of party A and were averaged over five samples.

Running time Figure 3.11 plots the end-to-end execution time of SYMPHONY and of Obliv-C on the benchmarks. On LAN under MPC (Yao), SYMPHONY's running time is 1.15× that of Obliv-C (per the geometric mean). Without MPC, SYMPHONY time is 2.4× that of Obliv-C. On WAN under MPC, SYMPHONY time is 0.85× that of Obliv-C. The maximum slowdown occurs in `edit-dist`, which uses dynamic programming and for Obliv-C is heavily optimized by GCC.

There are a two primary sources for SYMPHONY's overhead: First, SYMPHONY

supports arbitrary numbers of parties while Obliv-C supports only two. This is significant because SYMPHONY performs frequent runtime checks on the parties in scope. Since SYMPHONY supports an arbitrary number of possible parties, we represent the parties in scope as a set (implemented by a balanced tree data structure). Thus checks on principals are implemented by set operations. We could improve the efficiency of these runtime checks by implementing them using a bitset instead of a balanced tree. Obliv-C also performs certain checks on parties but, since only two parties are supported, these are implemented as simple integer equality checks.

Second, SYMPHONY is interpreted but Obliv-C is compiled. Interpretation imposes overhead, especially for programs involving loops. For example, a simple stress test which sums 1 million integers (in the clear) on a single party shows that SYMPHONY takes about 6 seconds where Obliv-C takes about 100 milliseconds. This stress test executes no runtime checks imposed by $\lambda_{\mathbf{MPC}}$, which suggests that the overhead is due to interpretation.

Since both SYMPHONY and Obliv-C are synchronous (i.e. they block when reading from the network), each non-local MPC operation imposes a RTT delay on the real execution time. If the implementations were asynchronous instead, the MPC operations and interpretation would execute in parallel. Instead of an additive delay, real execution time between non-local MPC operations would be the maximum of the interpretation time and RTT. For all but the fastest LAN networks, the RTT is $> 5ms$. We conjecture that the interpretive overhead of SYMPHONY is small enough that it is dominated entirely by the network latency for most deployments. If that is the case, real execution time between asynchronous SYMPHONY and Obliv-C would be indistinguishable.

Comparing the LAN and WAN benchmarks confirms that the language overhead imposed by SYMPHONY is dominated by the time it takes to perform network communication during a WAN deployment of MPC. We believe SYMPHONY is faster

than Obliv-C in the WAN setting due to its use of the EMP Integer interface, which uses the network more efficiently than the Bit interface used by Obliv-C's callback mechanism, and consequently the EMP backend for Obliv-C.

Generated circuit sizes As a second experiment, we instrumented the EMP backend to count the number of utilized AND and XOR gates. Counting gates is primarily a sanity check that ensures SYMPHONY is not erroneously introducing large numbers of unneeded gates. Table 3.2 tabulates the number of AND and XOR gates generated by SYMPHONY and by Obliv-C. The gate counts generated by SYMPHONY and Obliv-C are very similar, with differences caused by using the EMP Integer interface vs Obliv-C compiling to the Bit interface (as required by its callback mechanism. The optimizations performed by EMP's circuit compiler and Obliv-C's circuit compiler are similar, but not identical.

Overall, our experiments indicate that the language design itself does not impose significant overhead on either end-to-end execution time or generated circuit sizes. We leave a more sophisticated implementation which leverages compilation and compiler optimizations to future work.

3.7 Related Work

We compare SYMPHONY to existing MPC languages and frameworks, expanding the discussion from Section 3.1. Most of the data and analysis included in Table 3.3 and Table 3.4 comes from Hastings et al. [80]. Readers who want to learn even more about existing MPC languages should consult their paper.

Table 3.3 gives an overview of the most popular MPC languages and frameworks. The **Protocol** column indicates the protocol family supported by the language. GC and MC indicate support for garbled circuit and multiparty circuit protocols respectively. The Hy protocol family indicates that the language does not represent

Table 3.2: Gate counts (AND and XOR) of select benchmark programs. **Input Size** for Hamming Dist., Bio. Matching, DB Analytics, and Edit Dist. is the length of the input lists. For GCD, it is the maximum number of GCD iterations. Gate counts were collected by modifying EMP to record AND or XOR gate execution. Symphony uses EMP's Integer interface where applicable, OblivC uses EMP's Bit interface (compiling integer operations to circuits).

		OblivC		Symphony		Δ (OblivC - Symphony)	
Benchmark	Input Size	AND Gates	XOR Gates	AND Gates	XOR Gates	AND Gates	XOR Gates
Hamming Dist.	10000	1249875	3159595	950000	2550000	299875	609595
	20000	2499875	6319595	1900000	5100000	599875	1219595
	30000	3749875	9479595	2850000	7650000	899875	1829595
	40000	4999875	12639595	3800000	10200000	1199875	2439595
	50000	6249875	15799595	4750000	12750000	1499875	3049595
Bio. Matching	100	2617868	6353496	2675500	8007000	-57632	-1653504
	200	5235768	12706996	5351000	16014000	-115232	-3307004
	300	7853668	19060496	8026500	24021000	-172832	-4960504
	400	10471568	25413996	10702000	32028000	-230432	-6614004
	500	13089468	31767496	13377500	40035000	-288032	-8267504
DB Analytics	60	4609304	9569553	4732457	10425422	-123153	-855869
	70	6246968	12970159	6413597	14128242	-166629	-1158083
	80	8133020	16886701	8349937	18393262	-216917	-1506561
	90	10268008	21320663	10541477	23220482	-273469	-1899819
	100	12651370	26270229	12988217	28609902	-336847	-2339673
GCD	500	2360091	6735821	2302192	6910016	57899	-174195
	600	2832991	8085621	2762592	8291916	70399	-206295
	700	3305891	9435421	3222992	9673816	82899	-238395
	800	3778791	10785221	3683392	11055716	95399	-270495
	900	4251691	12135021	4143792	12437616	107899	-302595
Edit Dist.	50	780882	1704539	637372	1779607	143510	-75068
	80	2003142	4378936	1631872	4556407	371270	-177471
	110	3790602	8291824	3085372	8614807	705230	-322983
	140	6143262	13443100	4997872	13954807	1145390	-511707
	170	9061122	19832759	7369372	20576407	1691750	-743648

Table 3.3:

	Paper	Protocol	Parties	Semi-Honest	Malicious
EMP-toolkit	[176]	GC	2	●	●
Obliv-C	[181]	GC	2	●	○
ObliVM	[112]	GC	2	●	○
TinyGarble	[167]	GC	2	●	○
Wysteria	[143]	MC	2+	●	○
ABY	[54]	GC,MC	2	●	○
MOTION	[36]	GC,MC	2+	●	○
SCALE-MAMBA	[5]	Hy	2+	●	●
Sharemind	[31]	Hy	3	●	○
MPyC	[15]	MC	3+	●	○
PICCO	[188]	Hy	3+	●	○
SYMPHONY	-	GC,MC	2+	●	○

programs as circuits and instead represents certain important operations (e.g. squaring) with custom protocols. The **Parties** column indicates the number of supported parties, and the **Semi-Honest** and **Malicious** columns indicate the adversary model of the underlying protocols of the language.

In addition to the overview provided in Table 3.3, we also provide a more detailed view of the features provided by each language. The **Language** column indicates the programming language in which the framework is implemented. The **Custom**, **Extension**, and **Library** columns indicate if the MPC framework is a custom language, language extension, or library respectively. For example, Wysteria and SYMPHONY each provide a custom source language in which MPC programs are written. In contrast, EMP is a C++ library and OblivC extends the `gcc` compiler toolchain. Finally, the **Domain** column indicates whether the framework supports MPC over boolean (B) circuits, arithmetic (A) circuits, or both.

EMP and Obliv-C EMP-toolkit [176] and Obliv-C [181] are two-party, garbled circuit MPC frameworks written in C++ and C respectively. They are currently among the fastest frameworks available for two-party MPC. These frameworks are ideally suited for SIMD-style computation, since the "boilerplate" coordination for all

Table 3.4:

	Language	Custom	Extension	Library	Domain
EMP-toolkit	C++	○	○	●	B
Obliv-C	OCaml,C	○	C	○	B
ObliVM	Java	●	○	○	B
TinyGarble	C/C++	○	Verilog	○	B
Wysteria	OCaml	●	○	○	B
ABY	C++	○	○	●	B,A
MOTION	C++	○	○	●	B,A
SCALE-MAMBA	Python,C++	●	○	○	A
Sharemind	C/C++	●	○	○	A
MPyC	Python	○	○	●	A
PICCO	C/C++	○	C	○	A
SYMPHONY	Haskell,Rust	●	○	○	B,A

MPC programs (e.g. reading input, streaming gates) is handled inside the framework. EMP and Obliv-C both have support for reactive MPC, enabling intermediate values to be revealed for the purpose of optimization. These frameworks are well-suited for applications that do not require coordination, such as `hamming` or naive private set intersection, while requiring maximum efficiency. Applications like these do not benefit from SYMPHONY's expressiveness and flexible coordination. In contrast, EMP and Obliv-C would not be a good choice for applications that require various forms of coordination. One example is an application with $N \gg 2$ input parties that performs MPC over 2 compute parties before revealing to $K < N$ output parties. SYMPHONY makes this coordination simple while still allowing the compute parties to engage in a 2-party MPC protocol (e.g. by leveraging SYMPHONY's EMP backend).

TinyGarble and ObliVM TinyGarble [167] and ObliVM [112] are two-party, garbled circuit MPC languages written in C++ and Java respectively. TinyGarble takes a Verilog program and uses hardware circuit synthesis techniques to produce a boolean circuit. ObliVM takes a custom Java-like source language and compiles it to a boolean circuit. The ObliVM source language provides standard data types and operators (e.g. fixed-width integers with arithmetic and logical operators). It also provides built-in

support for private indexing using Circuit ORAM []. A distinguishing feature of both languages is their support for global, whole-program optimization. These languages produce small, highly-optimized circuits which are smaller than the circuits produced by frameworks like EMP and Obliv-C. All else being equal, these language are the most efficient option because the cost of circuit compilation is not incurred at runtime and the circuits produced are small. However, efficiency in practice may vary due to differences in protocol backend and engineering. As with EMP and Obliv-C, these languages are ideally suited for SIMD-style computation and are not an appropriate choice for applications that require coordination. Finally, practitioners should be aware that these are domain-specific languages which may lack certain features that are required for a particular application.

ABY and MOTION ABY [54] (2-party) and MOTION [36] (N-party) are mixed protocol MPC frameworks written in C++. Programs are written in C++ using classes to represent standard data types and overloaded operators for secure operations. In contrast to EMP and Obliv-C, these frameworks allow encrypted values to be converted between secret-shared and garbled representations to improve the efficiency of MPC applications that mix boolean and arithmetic computation. Both frameworks separate the circuit construction and circuit execution phases to achieve optimal round complexity for secret-sharing protocols. This is beneficial for performance, but it makes reactive MPC much more difficult since circuits must be constructed prior to evaluation. As a consequence, neither framework supports reactive MPC. These frameworks are optimal for SIMD-style applications that additionally contain a mix of Boolean and arithmetic operations and do not require optimization through reactive MPC. In contrast, Symphony supports non-SIMD, reactive MPC programs with mixed protocols.

SCALE-MAMBA SCALE-MAMBA [5] is an N-party, secret-sharing MPC language written in Python and C++. It is one of the most mature MPC frameworks available and supports both semi-honest and malicious security. Programs are written either in a Python or Rust DSL and compiled down to SCALE bytecode. The SCALE bytecode is designed to expose bytecode instructions that can be implemented efficiently using hybrid (Hy) protocols. For example, there is a dedicated instruction `SQUARE` for generating a pair of encrypted values (a, b) such that $b = a^2 \mod p$. SCALE-MAMBA is best suited to applications that require malicious security among $N > 2$ parties. While it does not provide any support for coordination tasks, it does support output of secret shares. This makes it possible, albeit unpleasant and error-prone, to implement coordination patterns by writing many SCALE-MAMBA programs and orchestrating them with a general-purpose language. SYMPHONY does not currently have support for any maliciously secure backends, but the protocol interface was designed with malicious security in mind.

Sharemind and MPyC Sharemind [31] (3-party) and MPyC [15] (N-party) are honest-majority, secret-sharing MPC frameworks. In contrast to the other languages discussed here, these frameworks operate in the honest-majority adversary model. For example, Sharemind assumes only 1 out of the 3 parties will be corrupted. In this setting, MPC can be computed much faster because the protocols need only rely on primitives that are information-theoretically secure (e.g. symmetric encryption). Sharemind has explicit support for input and output parties which delegate secure computation to the 3 compute parties running the Sharemind MPC framework protocol. Its source language, SecreC, supports reactive MPC through a `declassify` expression. Sharemind is ideally suited for MPC applications with large amounts of data that require support for delegation but no other forms of coordination. Languages like SYMPHONY should be preferred for applications that require more flexible

coordination or a stronger adversary model. The use cases for MPyC are similar, except that it does not have explicit support for delegation but does support $N > 3$ parties.

PICCO PICCO [188] is an 3+-party, secret-sharing MPC language which extends C. Like Obliv-C, it extends C with private data types and strives to support as much of C as possible. For example, it supports operations on private floats. Like Sharemind, it has explicit support for input, compute, and output parties. This provides support for a limited kind of delegation. One unique feature of PICCO is a lightweight syntax for denoting that loops over arrays of secrets should be executed concurrently. By exposing concurrency in source programs and leveraging a threshhold (honest-majority) protocol, PICCO was one of the fastest 3+ MPC framework at the time of its publication. We are not sure how its performance compares today, but it has been periodically updated since its release in 2013. PICCO is ideally suited for the same kind of applications as Sharemind while supporting more than 3 compute parties.

Viaduct Viaduct [2] compiles a Java-like language to secure distributed programs, which leverage cryptography, including MPC. Symphony supports richer coordination patterns than Viaduct (e.g., LWZ in Figure 3.1), due to its first-class principal sets, bundles, and par blocks. Viaduct's programming model is higher-level than Symphony; computation/communication patterns are synthesized based on security policies specified as IFC labels. It also supports cryptographic schemes beyond MPC (e.g., commitments, zero-knowledge proofs). Viaduct has no result corresponding to Symphony's soundness guarantee (Section 3.4), but has specific means to specify security policies, including those involving declassification and endorsement. Symphony essentially takes an "ideal world" approach, relying on the programmer to judge (using the ST semantics) that a program does not release too much.

Choreography Languages SYMPHONY's semantics of "generalized SIMD" bears resemblance to that of *choreography* languages [50, 51, 137, 124]. Choreographic programs are conceptually sequential, ensuring that `send` and `receive` operations are always matched up by combining them into a single expression. Pirouette [82] is a typed choreographic functional programming language which proves that the distributed deployment of well-typed programs is deadlock free by design. Pirouette is able to prove strong metatheoretic properties relating choreographies to their distributed deployment due to its static typing and static party annotations.

Chapter 4

λ_{Obliv}: A Language for Probabilistically Oblivious Computation

This chapter presents $\lambda_{\mathbf{Obliv}}$, a core language for oblivious computation, inspired by ObliVM. It extends a standard language with primitives for securely generating and using uniformly distributed random numbers. We prove that $\lambda_{\mathbf{Obliv}}$'s type system guarantees *probabilistic memory trace obliviousness* (PMTO), i.e., that the distribution of adversary-visible execution traces is independent of secret values. This property generalizes the deterministic MTO property enforced by Liu, Hicks, and Shi [108] and Liu et al. [111], 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 $\lambda_{\mathbf{Obliv}}$ is a design which achieves soundness without overly restricting or complicating the language.

$\lambda_{\mathbf{Obliv}}$'s type system uses *affine* types and a new mechanism that 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.) We prove that $\lambda_{\mathbf{Obliv}}$ enjoys PMTO by relating its semantics to a novel *mixed semantics* whose terms operate on distributions directly, which makes stating and proving the PMTO property much easier.

$\lambda_{\mathbf{Obliv}}$ is expressive enough to type check interesting algorithms. We present the implementation of a tree-based, non-recursive ORAM (NORAM) that type checks in a straightforward extension of $\lambda_{\mathbf{Obliv}}$; we have implemented a type checker for this extension. NORAM is a key component of state-of-the-art ORAM implementations [160, 169, 175] and other oblivious data structures [174], and to our knowledge ours is the first implementation automatically verified to be oblivious. We additionally show that *recursive* ORAM, built on NORAM, is possible but requires a few more advanced (but standard) language features we have not implemented, including region polymorphism, recursive and variant types, and existential quantification.

Finally, we have also experimented with implementing oblivious data structures using our NORAM implementation. We conclude by providing evidence that *oblivious stacks* (ostacks) don't satisfy PMTO due to the possibility of information leakage caused by overflow in the underlying NORAM.

4.1 Overview

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

4.1.1 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 [121]: The adversary knows the program being executed, and can observe during execution the PC, the contents of memory, and memory access patterns. Some *secret* memory contents may be encrypted (while *public* memory is not) but all addresses used to access memory are still visible.

Consider an untrusted cloud provider using a secure processor, like SGX [84]. Reads/writes from/to memory can be directly observed, but secret memory is encrypted (using a key kept by the processor). The pattern of accesses, timing information, and other system features (e.g., instruction cache misses) provide information about the PC. Another setting is secure multi-party computation (MPC) using secret shares [72]. Here, two parties simultaneously execute the same program (and thus know the program and program counter), but certain values—the input values from each party—are kept hidden from both using secret sharing.

By handling such a strong adversary, our techniques can also handle adversaries with fewer capabilities, such as those that can observe memory traffic but not the PC, or can make timing measurements but cannot observe the PC or memory.

4.1.2 Oblivious Execution

Our goal is to ensure *memory trace obliviousness (MTO)*, which is a kind of noninterference property [69, 150]. 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 addresses of public and secret values that are read and written. (Secret *values* are not visible in the trace.) Under this model, MTO means that running *low-equivalent* input states $\sigma_1; e_1$ and $\sigma_2; e_2$ will produce the exact same

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

(a) Leaky program

```
1  B[0] ← s0
2  B[1] ← s1
3  ...
4  let s = ...  (* secret *)
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 *)
7  let s' = xor(s,sk)
8  let r = B[castP(s')]
```

(c) Probabilistic MTO program

Figure 4.1: Code examples

memory trace, along with low-equivalent output states. Two states are low equivalent if they agree on the code and public values (but may differ on secret values). More formally, MTO states that if $\sigma_1; e_1 \sim \sigma_2; e_2$ and $\sigma_1; e_1 \longrightarrow^t \sigma_1'; e_1'$ then there exists $\sigma_2'; e_2'$ s.t. $\sigma_2; e_2 \longrightarrow^t \sigma_2'; e_2'$ and $\sigma_1'; e_1' \sim \sigma_2'; e_2'$, where $\sim$ denotes low-equivalence.

To illustrate how revealing addresses can leak information, consider the program in Figure 4.1(a). Here, we assume array B's contents are secret, and thus invisible to the adversary. Variables s0, s1, and s are secret (i.e., encrypted) inputs. The assignments on the first two lines are safe since we are just storing secret values in the secret array. The problem is on the last line, when the program uses s to index B. Since the adversary is able to see which address was used (in trace t), they can infer s.

The program in Figure 4.1(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.

4.1.3 Probabilistic Oblivious Execution

To improve performance while retaining security, the key is to employ randomness. In particular, the client can randomly generate and hold secret a key, using it to map logical addresses used by the program to physical addresses visible to the adversary. The program in Figure 4.1(c) illustrates the idea, hinting at the basic approach to implementing an ORAM. Rather than deterministically store s0 and s1 in positions 0 and 1 of B, respectively, the program scrambles their locations according to a coin flip, sk, generated by the call to **flip**, and not visible to the adversary. Using the **mux** on line 2, if sk is 1 then s0 and s1 will be copied to s0' and s1', respectively, but if sk is 0 then s0 and s1 will be swapped, with s0 going into s1' and s1 going into s0'. (The **castS** coercion on sk is a no-op, used by the type system; it will be explained in the next subsection.) Values s0' and s1' are then stored at positions 0 and 1, respectively, on lines 3 and 4. When the program later wishes to look up the value at logical index s, it must consult sk to retrieve the mapping. This is done via the **xor** on line 7. Then s' is used to index B and retrieve the value logically indicated by s.

In terms of memory accesses, this program is more efficient: It reads B only once, not twice. One can argue that more work is done overall, but as we will see in Section 4.4, this basic idea does scale up to build recursive ORAMs with access times of $O(log_c\ N)$ for some c (rather than $O(N)$).

This program is also secure: no matter the value of s, the adversary learns nothing from the address trace. Consider Figure 4.2 which tabulates the four possible traces (the memory indexes used to access B) depending on the possible values of s and sk. This table makes plain that our program is not *deterministically* MTO. Looking at column sk=0, we can see that a program that has s=0 may produce trace 0,1,0 while a program that uses s=1 may produce trace 0,1,1; MTO programs may not produce

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

Figure 4.2: Traces

different traces when using different secrets.

But this is not actually a problem. Assuming that $\mathsf{sk} = 0$ and $\mathsf{sk} = 1$ are equally likely, we can see that address traces 0,1,0 and 0,1,1 are also equally likely no matter whether $\mathsf{s} = 0$ or $\mathsf{s} = 1$. More specifically, if we assume the adversary's expectation for secret values is uniformly distributed, then after *conditioning* on knowledge of the third memory access, the adversary's expectation for the secret remains unchanged, and thus nothing is learned about s. This probabilistic model of adversary knowledge is captured by a *probabilistic* variant of MTO. In particular, the probability of any particular trace event t emitted by two low-equivalent programs should be the same for both programs, and the resulting programs should also be low-equivalent. More formally: If $\sigma_1; e_1 \sim \sigma_2; e_2$ then $\Pr[\sigma_1; e_1 \longrightarrow^t \sigma_1'; e_1'] = q$ implies $\Pr[\sigma_2; e_2 \longrightarrow^t \sigma_2'; e_2'] = q$ and $\sigma_1'; e_1' \sim \sigma_2'; e_2'$.

4.1.4 $\lambda_{\mathbf{Obliv}}$: Obliviousness by Typing

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

Affinity In $\lambda_{\mathbf{Obliv}}$, public and secret bits are given types $\mathtt{bit}_{\mathrm{P}}$ and $\mathtt{bit}_{\mathrm{S}}$ respectively, and coin flips are given type $\mathtt{flip}$. Our formalism uses bits for simplicity; it is easy to generalize to (random fixed-width) integers, which is done in our implementation.

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

(a) Leak by multiple revelation

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

(b) Leak due to probabilistic dependence

Figure 4.3: Example leaky programs (precluded by $\lambda_{\mathbf{Obliv}}$ type system)

Values of `flip` type are, like secret bits of type $\texttt{bit}_S$, invisible to the adversary. But a `flip` can be revealed by using $\texttt{cast}_P$ to convert it to a public bit, as is done on line 8 of Figure 4.1(c) to perform a (publicly visible) array index operation.

The type system aims to ensure that a `flip` value is always uniformly distributed when it is revealed. The uniformity requirement implies that each flip should be revealed *at most once*. Why? Because the second time a flip is revealed, its distribution is conditioned on prior revelations, meaning the each outcome is no longer equally likely. To see how this situation could end up leaking secret information, consider the example in Figure 4.3(a). Lines 1–3 in this code are safe: we generate two coin flips that are invisible to the adversary, and then store one of them in `sz` depending on whether the secret `s` is 1 or not. Revealing `sz` at line 3 is safe: regardless of whether `sz` contains the contents of `sx` or `sy`, the fact that both are uniformly distributed means that whatever is revealed, nothing can be learned about `s`. However, revealing `sx` on line 4, after having revealed `sz`, is not safe. This is because seeing two ones or two zeroes in a row is more likely when `sz` is `sx`, which happens when `s` is one. So this program violates PMTO.

To prevent this problem, $\lambda_{\mathbf{Obliv}}$'s type system treats values of type `flip` affinely, meaning that each can be used at most once. The read of `sx` on line 2 consumes that variable, so it cannot be used again on the problematic line 4. Likewise, flip variable `sk` is consumed when passed to **xor** on line 7 of Figure 4.1(c), and `s'` is consumed when revealed on line 8.

Unfortunately, a purely affine treatment of flips would preclude useful algorithms.

In particular, notice that line 2 of Figure 4.1(c) uses sk as the guard of a **mux**. If doing so consumed sk, line 7's use of sk would fail to type check. To avoid this problem, $\lambda_{\textbf{Obliv}}$ relaxes the affinity constraint on flips passed to **castS**. In effect, programs can make many secret $\texttt{bit}_{\texttt{S}}$ copies of a flip, and compute with them, but only the original flip can ultimately be revealed.

It turns out that this relaxed treatment of affinity is insufficient to ensure PMTO. The reason is that we can now use non-affine copies of a coin to make a flip's distribution non-uniform when it is revealed. To see how, consider the code in Figure 4.3(b). This code flips two coins, and then uses the **mux** to store the first coin flip, sx, in sk if sx is 1, else to store the second coin flip there. Now sk is more likely to be 1 than not: $\Pr[\mathsf{sk} = 1] = \frac{3}{4}$ while $\Pr[\mathsf{sk} = 0] = \frac{1}{4}$. 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 4.1(c) with the first two lines above: when the program looks up B[**castP**(s')] on line 8, if the adversary observes 1 for the address, it is more likely that s is 0, and vice versa if the adversary observes 1. Notice that we have not violated affinity here: no coin flip has been used more than once (other than uses of **castS** which side-step affinity tracking). The problematic correlation in Figure 4.3(b) is incorrectly allowed by ObliVM [112], and is the root of its unsoundness.

Probability regions $\lambda_{\textbf{Obliv}}$'s type system addresses the problem of probabilistic correlations leading to non-uniform distributions using a novel construct we call *probability regions*, which are static names that represent sets of coin flips, reminiscent of a points-to location in alias analysis [57]. We have elided the region name in our examples so far, but normally programmers should write $\texttt{flip}^{\rho}()$ for flipping a coin in region ρ, which then has type $\texttt{flip}^{\rho}$. Bits derived from flips via **castS** carry the region of the original flip, so bit types also include a region ρ.

Regions form a partial order, and the type system enforces an invariant that each flip labeled with region ρ is probabilistically independent of all bits derived from flips at regions ρ' when $\rho' \sqsubset \rho$. Then, the type system will prevent problematic correlations arising among bits and flips, in particular via the **mux** and **xor** operations, in a way that could threaten uniformity. We can see regions at work in the problematic example above: the region of the secret bit **castS**(sx) is the same region as sx, since **castS**(sx) was derived from sx. As such, there is no assurance of probabilistic independence between the guard and the branch; indeed, when conditioning on **castS**(sx) to return sx, the output will *not* be uniform. On the other hand, if the guard of a **mux** is a bit in region ρ and its branches are flips in region ρ' where $\rho \sqsubset \rho'$, then the guard is derived from a flip that is sure to be independent of the branches, so the uniformity of the output is not threatened. This kind of provable independence is a critical piece of our Tree ORAM implementation in Section 4.4.

4.2 Formalism

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

4.2.1 Syntax

Figure 5.4 shows the syntax for $\lambda_{\textbf{Obliv}}$. The term language is expressions e. The set of values v is comprised of (1) base values such as variables x (included to enable a substitution-based semantics) and recursive function definitions $\texttt{fun}_y(x{:}\tau).e$ where the function body may refer to itself using variable y; and (2) connectives from the expression language e which identify a subset of expressions which are also values, such as pairs $\langle v, v\rangle$ with type $\tau \times \tau$.

Expressions also include bit literals b_ℓ (of type $\texttt{bit}^{\perp}_{\ell}$) which are either `0` or `1` and

ℓ	$\in$	label	$::=$	$\mathrm{P} \mid \mathrm{S}$	public and secret
				(*where* $\mathrm{P} \sqsubset \mathrm{S}$)	security labels
ρ	$\in$	R	$::=$	$\ldots$	probability region
b	$\in$	$\mathbb{B}$	$::=$	$\mathtt{0} \mid \mathtt{1}$	bits
x, y	$\in$	var	$::=$	$\ldots$	variables
v	$\in$	val	$::=$	x	variable values
			$\mid$	$\mathtt{fun}_y(x{:}\tau).e$	function values
			$\mid$	$\langle v, v \rangle$	tuple values
τ	$\in$	type	$::=$	$\mathtt{bit}_\ell^\rho$	non-random bit
			$\mid$	$\mathtt{flip}^\rho$	secret uniform bit
			$\mid$	$\mathtt{ref}(\tau)$	reference
			$\mid$	$\tau \times \tau$	tuple
			$\mid$	$\tau \to \tau$	function
e	$\in$	exp	$::=$	v	value expressions
			$\mid$	b_ℓ	bit literal
			$\mid$	$\mathtt{flip}^\rho()$	coin flip in region
			$\mid$	$\mathtt{cast}_\ell(v)$	cast flip to bit
			$\mid$	$\mathtt{mux}(e, e, e)$	atomic conditional
			$\mid$	$\mathtt{xor}(e, e)$	bit xor
			$\mid$	$\mathtt{if}(e)\{e\}\{e\}$	branch conditional
			$\mid$	$\mathtt{ref}(e)$	reference creation
			$\mid$	$\mathtt{read}(e)$	reference read
			$\mid$	$\mathtt{write}(e, e)$	reference write
			$\mid$	$\langle e, e \rangle$	tuple creation
			$\mid$	$\mathtt{let}\ x = e\ \mathtt{in}\ e$	variable binding
			$\mid$	$\mathtt{let}\ x, y = e\ \mathtt{in}\ e$	tuple elimination
			$\mid$	$e(e)$	fun. application

Figure 4.4: $\lambda_{\mathbf{Obliv}}$ Syntax (source programs)

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

The expression $\texttt{mux}(e_1, e_2, e_3)$ unconditionally evaluates e_2 and e_3 and returns their values as a pair in the given order if e_1 evaluates to `1`, or in the opposite order if it evaluates to `0`. This operation is critical for obliviousness because it is atomic. By contrast, normal conditionals $\texttt{if}(e_1)\{e_2\}\{e_3\}$ evaluate either e_2 or e_3 depending on e_1, never both, so the branch taken is evident from the trace. The components of tuples e constructed as $\langle e_1, e_2\rangle$ can be accessed via $\texttt{let}\ x_1, x_2 = e\ \texttt{in}\ \ldots$ $\lambda_{\mathbf{Obliv}}$ also has normal let binding, function application, and means to manipulate mutable reference cells.

$\lambda_{\mathbf{Obliv}}$ 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. Other features can be encoded in these, e.g., general numbers and operators on them can be encoded as tuples of bits, and arrays can be encoded as tuples of references (read/written using (nested) conditionals). Our prototype interpreter implements these things directly.

4.2.2 Semantics

Figure 4.5 presents a monadic, probabilistic small-step semantics for $\lambda_{\mathbf{Obliv}}$ programs. The top of the figure contains some new and extended syntax. Values (and, by

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

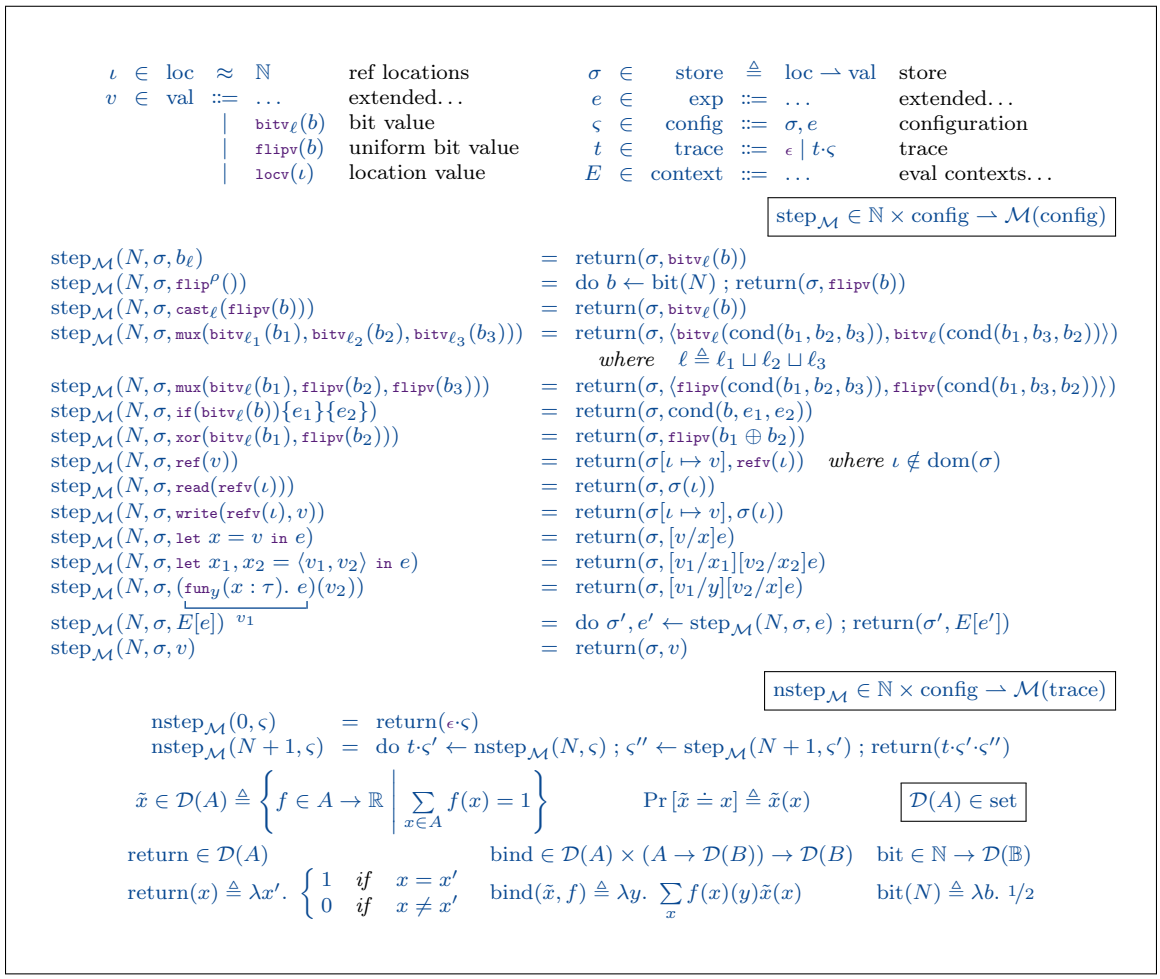


Figure 4.5: $\lambda_{\textbf{Obliv}}$ Semantics

extension, expressions) are extended with forms for bit values $\texttt{bitv}_\ell(b)$, flip values $\texttt{flipv}(b)$, and reference locations $\texttt{locv}(\iota)$; these do not appear in source programs. Stores σ map locations to values. Stores are paired with expressions to form *configurations* ς. A sequence of configurations arising during an evaluation is collected in a *trace* t. We define evaluation contexts E (not shown) in the style of Felleisen and Hieb [61] to enforce a left-to-right, call-by-value evaluation strategy.

The semantics is defined using an abstract probability monad $\mathcal{M}$. Below the semantics we define the standard "denotational" discrete probability monad $\mathcal{D}$ [68, 140]. The *standard* semantics for our language occurs when $\mathcal{M} = \mathcal{D}$, and we leave $\mathcal{M}$ a parameter so we can instantiate the semantics to a new monad in the next section.

In the probability monad $\mathcal{D}$, the return operation constructs a point distribution, and the bind operation encodes the law of total probability, i.e., constructs a marginal distribution from a conditional one. We only use proper distributions in the sense that the combined mass of all elements sums to 1. We do not denote possibly non-terminating programs directly into the monad, and therefore do not require the use of computable distributions [87] or sub-probability distributions [122]—we use the monad only to denote distributions of configurations which occur after a finite number of small-step transitions, which is total.

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

Moving to the first case, we see that literals b_ℓ evaluate in one step to bit values.

A $\texttt{flip}^{\rho}()$ expression evaluates to either $\texttt{flipv}(\texttt{I})$ or $\texttt{flipv}(\texttt{O})$ as determined by $\text{bit}(N)$, which for the monad $\mathcal{D}$ yields $1/2$ probability for each outcome. (The monad $\mathcal{D}$ does not use the N parameter in its definition of $\text{bit}(N)$, but a later monad will.) The $\texttt{cast}_{\ell}$ case converts a flip to a similarly-labeled bit value. The next few cases use the three-argument metafunction $\text{cond}(b, X, Y)$, which returns X if b is $\texttt{I}$, and Y otherwise. The two **mux** cases operate in a similar way: they return the second two arguments of the **mux** in order when the first argument is $\texttt{bitv}_{\ell}(\texttt{I})$, and in reverse order when it is $\texttt{bitv}_{\ell}(\texttt{O})$. The security label of the result is the join of the labels of all elements in involved. (This is not needed for flip values, since these are always fixed to be secret.) The case for **if** also uses cond in the expected manner. The case for **xor** permits xor-ing a bit with a flip, returning a flip.

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

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

4.2.3 Type System

Figure 4.6 defines the type system for $\lambda_{\textbf{Obliv}}$ source programs as rules for judgment $\Gamma \vdash e : \tau \;;\; \Gamma'$, which states that under type environment Γ expression e has type τ, and yields residual type environment Γ'. We discuss typing configurations, including non-source program values, in the next section. Type environments map variables to either types τ or inaccessibility tags $\bullet$, which are used to enforce affinity of flips. We discuss the three key features of the type system—affinity, probability regions, and information flow control—in turn.

$$\overset{\bullet}{\tau} \in \overset{\bullet}{\text{type}} ::= \tau \mid \bullet \quad (\textit{where } \tau \sqsubset \bullet) \qquad \Gamma \in \text{tcxt} \triangleq \textit{var} \rightharpoonup \overset{\bullet}{\text{type}}$$
$$\kappa \in \text{kind} ::= \mathtt{U} \mid \mathtt{A} \quad (\textit{where } \mathtt{U} \sqsubset \mathtt{A}) \qquad (\Gamma_1 \sqcup \Gamma_2)(x) \triangleq \Gamma_1(x) \sqcup \Gamma_2(x)$$

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

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

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

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

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

Bit
$$\frac{}{\Gamma \vdash b_\ell : \mathtt{bit}^{\perp}_{\ell} \dashv \Gamma}$$

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

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

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

If
$$\frac{\Gamma \vdash e : \mathtt{bit}^{\perp}_{\mathrm{P}} \dashv \Gamma' \quad \Gamma' \vdash e_1 : \tau \dashv \Gamma_1'' \quad \Gamma' \vdash e_2 : \tau \dashv \Gamma_2''}{\Gamma \vdash \mathtt{if}(e)\{e_1\}\{e_2\} : \tau \dashv \Gamma_1'' \sqcup \Gamma_2''}$$

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

Mux-Flip
$$\frac{\Gamma \vdash e_1 : \mathtt{bit}^{\rho_1}_{\ell_1} \dashv \Gamma' \quad \rho_1 \sqsubset \rho_2 \quad \Gamma' \vdash e_2 : \mathtt{flip}^{\rho_2} \dashv \Gamma'' \quad \rho_1 \sqsubset \rho_3 \quad \Gamma'' \vdash e_3 : \mathtt{flip}^{\rho_3} \dashv \Gamma''' \quad \rho = \rho_2 \sqcap \rho_3}{\Gamma \vdash \mathtt{mux}(e_1, e_2, e_3) : \mathtt{flip}^{\rho} \times \mathtt{flip}^{\rho} \dashv \Gamma'''}$$

Xor-Flip
$$\frac{\Gamma \vdash e_1 : \mathtt{bit}^{\rho_1}_{\ell_1} \dashv \Gamma' \quad \Gamma' \vdash e_2 : \mathtt{flip}^{\rho_2} \dashv \Gamma'' \quad \rho_1 \sqsubset \rho_2}{\Gamma \vdash \mathtt{xor}(e_1, e_2) : \mathtt{flip}^{\rho_2} \dashv \Gamma''}$$

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

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

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

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

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

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

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

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

Figure 4.6: $\lambda_{\mathbf{Obliv}}$ Type System (source programs)

Affinity To enforce non-duplicability, when an affine variable is used by the program, its type is removed from the residual environment. Figure 4.6 defines kinding metafunction $\mathcal{K}$ that assigns a type either the kind universal $\mathtt{U}$ (freely duplicatable) or affine $\mathtt{A}$ (non-duplicatable). Bits, functions, and references (but not their contents, necessarily) are always universal, and flips are always affine. A pair is considered affine if either of its components is. Rule VarU in Figure 4.6 types universally-kinded variables; the output environment Γ is the same as the input environment. Rule VarA types an affine variable by marking it $\bullet$ in the output environment. This rule is sufficient to rule out the first problematic example in Section 4.1.4.

Rules Cast-S and Cast-P permit converting flips to bits via the $\mathtt{cast}_{\mathrm{S}}$ and $\mathtt{cast}_{\mathrm{P}}$ coercions, respectively. The first converts a $\mathtt{flip}^{\rho}$ to a $\mathtt{bit}_{\mathrm{S}}^{\rho}$ and does *not* make its argument inaccessible (it returns the original Γ) while the second converts to a $\mathtt{bit}_{\mathrm{P}}^{\perp}$ and does make it inaccessible (returning Γ'). The type system is enforcing that any random number is made adversary-visible at most once; secret copies are allowed because they are never revealed.

References may contain affine values, but references themselves are universal. Rather than track the affinity of aliased contents specifically, the Read rule disallows reading out of a reference cell whose contents are affine. Since the write operation returns the *old* contents of the cell, programs can see the existing contents of any reference by first writing in a valid replacement [12].

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

Finally, note that different variables could be made inaccessible in different branches of a conditional, so If types each branch in the same initial context, but then joins

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

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

The rules also aim to prevent *implicit* information flows. A typical static information flow type system [150] would require the type of the conditional's guard to be less secret than the type of what it returns; e.g., the guard's type could be $\texttt{bit}^{\rho}_{\mathrm{S}}$ but only if the final type τ is secret too. However, in $\lambda_{\textbf{Obliv}}$ we must be more restrictive: rule If requires the guard to be public since the adversary-visible execution trace reveals which branch is taken, and thus the truth of the guard. Branching on secrets must be done via **mux**. Notice that rule Mux-Bit sets the label ℓ of the each element of the returned pair to be the join of the labels on the guard and the remaining components. As such, if the guard was secret, then the returned results will be. The Mux-Flip rule always returns flips, which are invisible to the adversary, so the guard can be secret or public.

Probability regions. A probability region ρ appears on both `bit` and `flip` types. The region is a static name for a collection of flip values and secret bit values that may be derived from them. A flip value is associated with a region ρ when it is created, per

rule Flip. Rule Cast-S ascribes the region ρ from the input $\texttt{flip}^{\rho}$ to the output type $\texttt{bit}^{\rho}_{\mathrm{S}}$, tracking the flip value(s) from which the secret bit value was possibly derived. Per rule Bit, bit literals have probability region $\bot$, as do public bits produced by $\texttt{cast}_{\mathrm{P}}$, per rule Cast-P.

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

Consider the Mux-Flip rule. If a secret bit is typed at region ρ_1 and a flip value at region ρ_2, and $\rho_1 \not\sqsubset \rho_2$, then it may be that the values are correlated, and a $\texttt{mux}$ involving the values may produce flips that are non-uniform. The Mux-Bit rule returns outputs whose region is the *join* of the regions of the branches, and the Mux-Flip rule returns outputs whose region is the *meet* of the regions of the branches. This indicates that the result of the $\texttt{mux}$ is only independent of $\texttt{flip}$s that are jointly independent of ρ_2 and ρ_3. The use of join for $\texttt{bit}$s and meet for $\texttt{flip}$s follows from the semantic property of the ordering $\sqsubset$.

Because freshly generated random bits are always independent of each other, the programmer is free to choose any regions when generating them via $\texttt{flip}^{\rho}()$ expressions. However, once chosen, the ordering establishes an invariant which constrains the order in which mux operations can occur subsequently in the program. Requiring strict region ordering for mux operations is enough to reject the example from the end of Section 4.1.4, as it could produce a non-uniform coin $\texttt{sk}$. We recast the example below, labeled (a), using regions $\rho_1 \sqsubset \rho_2$.

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

(a) Incorrect example

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

(b) Correct example

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

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

We might be tempted not to order regions but instead maintain an invariant that flips and bits in distinct regions are independent. This turns out to not work. While at the outset a fresh flip value is independent of all other values in the context of the program, the region ordering is needed to ensure that mux operations will only occur in "one direction." E.g., if two fresh flip values are created $x = \texttt{flip}^{\rho_1}()$ and $y = \texttt{flip}^{\rho_2}$, it is true that x and y are mutually independent. Thus it would seem reasonable that $\texttt{mux}(\texttt{cast}_S(x), y, \ldots)$ and $\texttt{mux}(\texttt{cast}_S(y), x, \ldots)$ should both be well typed. While they are both safe in isolation, the combination is problematic. Consider the results of each mux—they are both flip values, and they are both valid to reveal using $\texttt{cast}_P$ individually. However, the resulting values are correlated (revealing one tells you information about the distribution of the other), which violates the uniformity guarantee of all $\texttt{cast}_P$ results. By ordering the regions, we are essentially promising to only allow mux operations like this in one direction but not the other, and therefore

uniformity is never violated for revealed flip values. For example, by requiring $\rho_1 \sqsubset \rho_2$ we allow the first `mux` above but not the second.

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

4.3 Probabilistic Memory Trace Obliviousness

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

4.3.1 What is PMTO?

Figure 4.7 presents a model obs of the adversary's view of a computation as a new class of values, expressions and traces that "hide" sub-expressions considered to be secret (written $\bullet$). Secret bit expressions, secret bit values, and secret flip values all map to $\bullet$. Compound values, expressions, stores, traces etc. call obs in recursive positions as expected.

Probabilistic memory trace obliviousness (PMTO), stated formally below, holds when observationally equivalent configurations induce distributions of traces that are themselves observationally equivalent after N steps, for any N.[2]

Proposition 4.3.1 (Probabilistic Memory Trace Obliviousness (PMTO))**.**

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

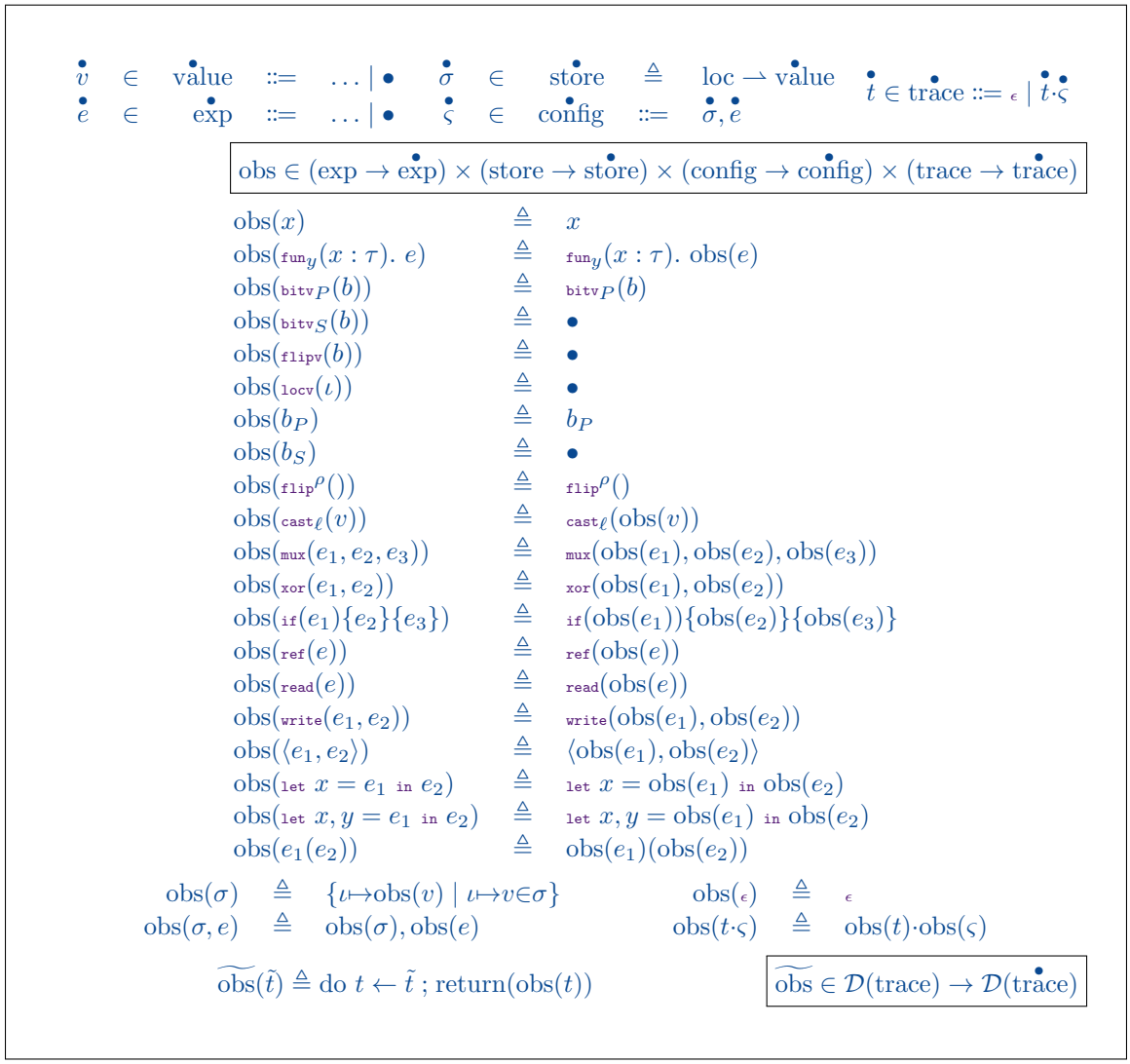


Figure 4.7: Adversary observability

If: e_1 *and* e_2 *are closed source expressions,* $\vdash e_1 : \tau$, $\vdash e_2 : \tau$ *and* $\mathrm{obs}(e_1) = \mathrm{obs}(e_2)$

Then: (1) $\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_1)$ *and* $\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_2)$ *are defined*

And: (2) $\widetilde{\mathrm{obs}}(\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_1)) = \widetilde{\mathrm{obs}}(\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_2))$.

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

4.3.2 Proof Approach

The remainder of this section works through our proof of PMTO (Theorem 4.3.1) which we complete in the following steps: (1) we develop a new probability monad called "intensional distributions" which simplifies reasoning about conditional independence between probabilistic values (§4.3.4); (2) we define an alternative syntax, semantics and type system for $\lambda_{\mathbf{Obliv}}$ programs called the "mixed semantics" which uses intensional distributions to simplify inductive reasoning about the adversary's view of probabilistic secret values (§4.3.3, §4.3.5); (3) we show that evaluation in the mixed semantics corresponds exactly with the ground truth semantics through simulation lemmas; (4) we prove that key invariants about probabilistic values are ensured by well-typed mixed terms, and that terms remain well-typed throughout evaluation—this establishes PMTO for the mixed semantics; and (5) we demonstrate PMTO for the ground truth semantics as a consequence of

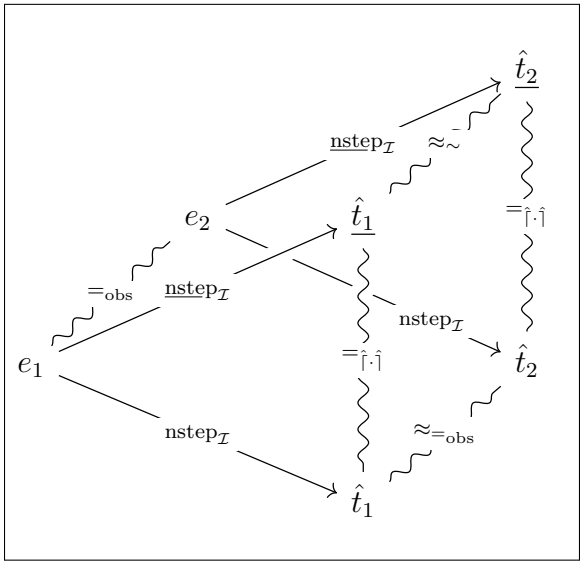


Figure 4.8: Proof Approach as a Diagram

lemmas established in steps (3–4) and a soundness lemma relating equivalent distributions of mixed terms to adversary-equivalent distributions of standard terms.

In Figure 4.8 we summarize the structure of this proof approach in a diagram. On the left are two programs e_1 and e_2 which are equal modulo adversary observation $=_{\text{obs}}$, which translates to $\text{obs}(e_1) = \text{obs}(e_2)$ as sketched in Proposition 4.3.1, and means e_1 and e_2 agree on public values and program structure but may differ in secrets. The rightward moving arrows represent running each program in either the ground truth semantics $\text{step}_{\mathcal{I}}$—the same semantics from Figure 4.5 but instantiated with the intensional distribution monad $\mathcal{I}$—and the mixed semantics $\underline{\text{step}}_{\mathcal{I}}$. Each of these executions result in intensional distributions of standard and mixed traces, respectively. In step (3) above we prove Lemma 4.3.0.1 to show these distributions are equivalent according to $=_{\lceil\cdot\rceil}$ which uses $\lceil\cdot\rceil$ to project distributions of mixed traces to distributions of standard traces. In step (4) above we prove Lemma 4.3.0.4 to establish PMTO for the mixed semantics; i.e., that the resulting distributions of mixed traces are equivalent modulo an underlying low-equivalence relation $\approx_{\sim}$. In step (5) we prove Lemma 4.3.0.5, which combines results from (3–4) to establish PMTO for the standard semantics (instantiated with $\mathcal{I}$)—the resulting distributions of standard traces are equivalent modulo equality of adversary observations, notated $\approx_{=_{\text{obs}}}$. The last step of PMTO (Theorem 4.3.1) is not shown: Lemma 4.3.0.2 proves via simulation that the intensional distribution monad $\mathcal{I}$ corresponds with the usual denotational probability monad presented in Section 5.2.

4.3.3 Mixed Semantics

An intuitive approach to proving Proposition 4.3.1 is to prove that a single-step version of it holds for $\text{step}_{\mathcal{D}}$, and then use that fact in an inductive proof over $\text{nstep}_{\mathcal{D}}$. Unfortunately, proving the single-step version quickly runs into trouble. Consider a source program $\texttt{castP}(\texttt{flip}^{\rho}())$ which steps to each of the expressions $\texttt{castP}(\texttt{flipv}(\texttt{I}))$ and

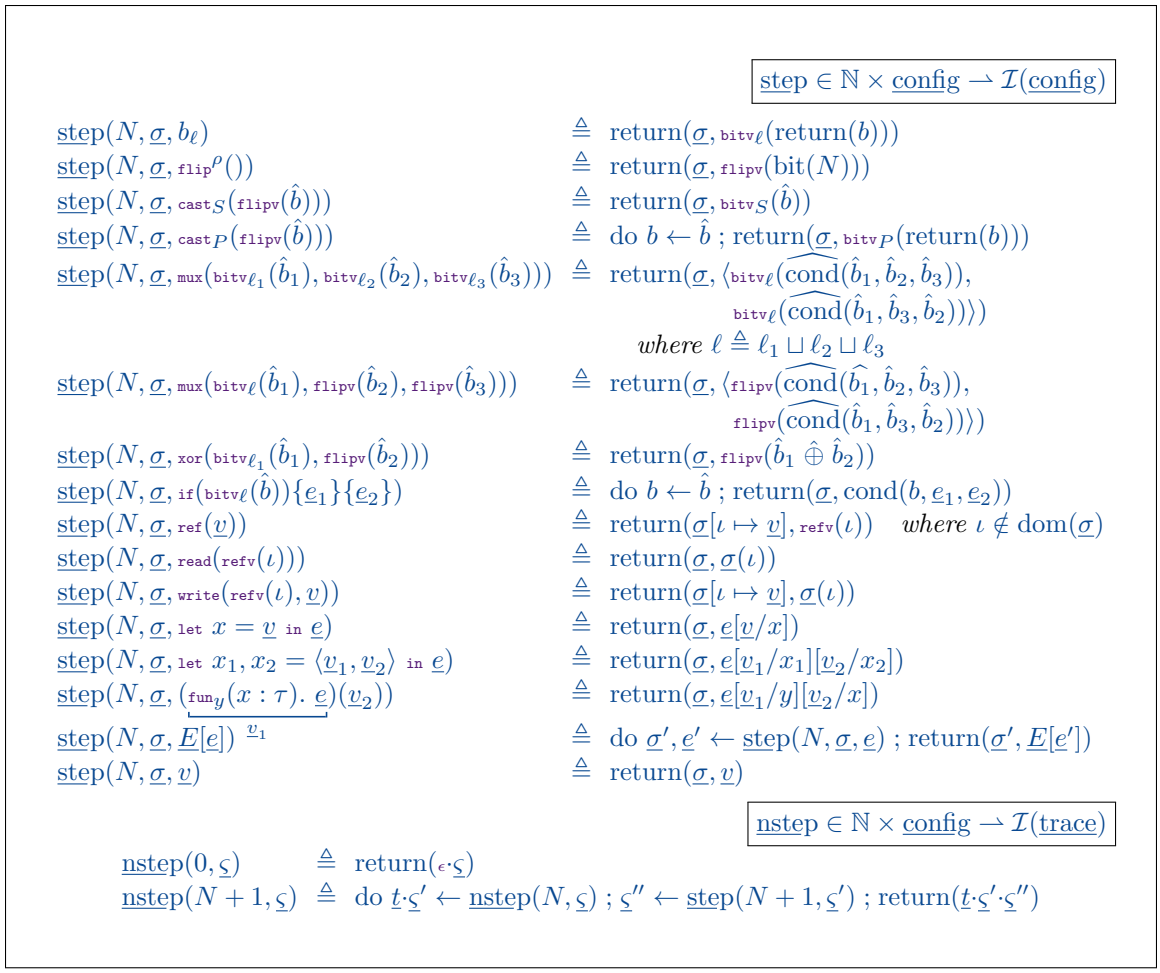


Figure 4.9: Mixed Language Semantics, where $\hat{b} \in \mathcal{I}(\mathbb{B})$ is a distributional bit value (see text)

$\texttt{cast}_P(\texttt{flipv}(\texttt{0}))$ with probability $1/2$. These expressions are observationally equivalent—the adversary's view of each is $\texttt{cast}_P(\bullet)$. For single-step PMTO to be satisfied, each of these terms must step to an equivalent distribution. Unfortunately, they do not: The first produces a point distribution of the expression $\texttt{bitv}_P(\texttt{1})$ and the second produces a point distribution of the expression $\texttt{bitv}_P(\texttt{0})$, which are not observationally the same.

To address this problem, we define an alternative *mixed* semantics which embeds *distributional bit values* directly into (single) traces. Instead of the semantics of $\texttt{flip}^\rho()$ producing two possible outcomes, in the mixed semantics it produces just one: a single distributional value $\texttt{flipv}(\hat{b})$ where the $\hat{b}$ represents either $\texttt{1}$ or $\texttt{0}$ with equal probability.

Doing this is like treating $\texttt{flip}^{\rho}()$ expressions lazily, and lines up (mixed) traces with the adversary's view $\bullet$.

The mixed semantics amends the syntax of $\texttt{flipv}$ and $\texttt{bitv}_{\ell}$ to be distributional (i.e., they contain $\hat{b}$ rather than just b). Other values from the standard semantics' syntax (top of Figure 4.5) are unchanged. As such, a distribution of pairs of bit values (say) is represented as pair of distributional bit values. To allow values inside the pair to be correlated, we represent them using what we call *intensional distributions*—intensional distributions are written $\mathcal{I}(A)$ and discussed in the next subsection.

The mixed semantics is shown in Figure 4.9. The mixed semantics step function $\underline{\text{step}}(N, \underline{\sigma}, \underline{e})$ maps a configuration, $\underline{\varsigma} \triangleq \underline{\sigma}, \underline{e}$ to an intensional distribution of configurations $\mathcal{I}(\underline{\text{config}})$. Mixed semantics expressions (and values, etc.) are underlined to distinguish them from the standard semantics, and operations on distributional values are hatted.

Most of the cases for the mixed semantics are structurally the same as the standard semantics. The key differences are the handling of $\texttt{flip}^{\rho}()$ and $\texttt{cast}_{\ell}(\underline{v})$. For the first, the standard semantics samples from the fresh uniform distribution immediately, while the mixed semantics produces a single uniform distributional value. This distributional value is sampled at the evaluation of $\texttt{cast}_{P}$, which matches the adversary's view.

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

4.3.4 Capturing Correlations with Intensional Distributions

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

Example Consider the program $\texttt{let}\ x = \texttt{flip}^{\rho}()\ \texttt{in}\ \langle \texttt{cast}_{\texttt{P}}(x), \texttt{cast}_{\texttt{P}}(x)\rangle$.[3] After two evaluation steps in the standard semantics, the program will be reduced to either $\langle \texttt{cast}_{\texttt{P}}(\texttt{flipv}(\texttt{I})), \texttt{cast}_{\texttt{P}}(\texttt{flip}$ or $\langle \texttt{cast}_{\texttt{P}}(\texttt{flipv}(\texttt{O})), \texttt{cast}_{\texttt{P}}(\texttt{flipv}(\texttt{O}))\rangle$, with equal probability. The standard rules for $\texttt{cast}_{\texttt{P}}$ would then yield (equally likely) $\langle \texttt{bitv}_{\texttt{P}}(\texttt{I}), \texttt{bitv}_{\texttt{P}}(\texttt{I})\rangle$ and $\langle \texttt{bitv}_{\texttt{P}}(\texttt{O}), \texttt{bitv}_{\texttt{P}}(\texttt{O})\rangle$. In the mixed semantics this program will evaluate in two steps to $\langle \texttt{cast}_{\texttt{P}}(\texttt{flipv}(\hat{b})), \texttt{cast}_{\texttt{P}}(\texttt{flipv}(\hat{b}))\rangle$ where $\hat{b}$ is a distributional value. At this point, the mixed semantics rule for $\texttt{cast}_{\texttt{P}}$ uses monadic bind to sample $\hat{b}$ to yield some b (which is either $\texttt{I}$ or $\texttt{O}$) and return it as a point distribution. The semantics needs to "remember" the bit chosen for the first $\texttt{cast}_{\texttt{P}}$ so that when it samples the second, the same bit is returned. Sampling independently would yield incorrect outcomes such as $\langle \texttt{bitv}_{\texttt{P}}(\texttt{O}), \texttt{bitv}_{\texttt{P}}(\texttt{I})\rangle$.

Intensional distributions As shown in the upper left of Figure 4.10, an intensional distribution $\mathcal{I}(A)$ over a set A is a binary tree with elements a of A at the leaves. It represents a distribution as a function from input entropy—a sequence of coin flips—to a result in A. Each node $\langle \hat{x}_1\ \hat{x}_2 \rangle$ in the tree represents two sets of worlds determined by the result of a coin flip: the left side $\hat{x}_1$ defines the worlds in which the coin was heads, and the right side $\hat{x}_2$ defines those in which it was tails. Each level of the tree represents a distinct coin flip, with the earliest coin flip at the root, and later coin flips at lower levels. The height of a tree represents an upper bound on the number of coin flips upon which a distribution's values depends. Each path through the tree is a possible world.

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

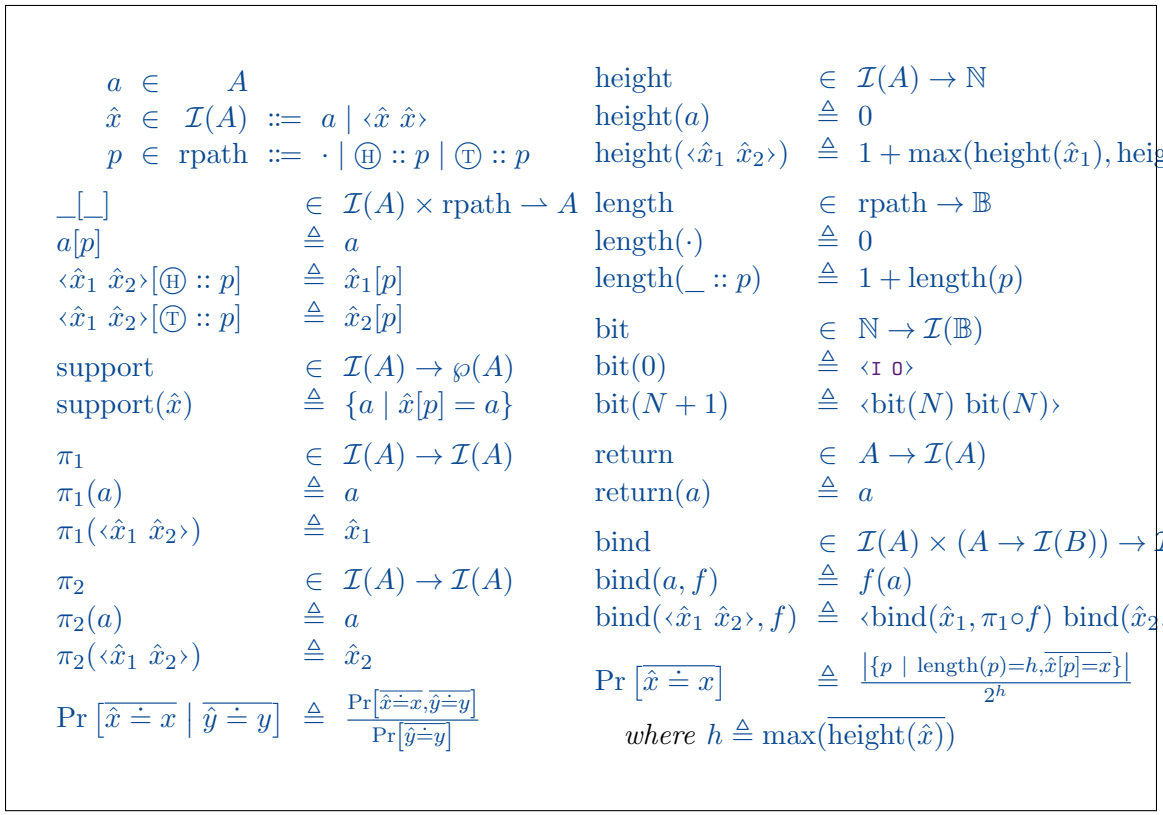


Figure 4.10: Intensional Distributions

For example, ‹‹3 4› ‹3 5›› is an intensional distribution of numbers in a scenario where two coins have been flipped. There are four possible worlds. ‹3 4› is the world where the 0th coin came up heads. 3 is the outcome in the world where both coins came up heads, while 4 is the outcome where the 0th coin was heads but the 1th coin was tails. ‹3 5› is the world where the 0th coin came up tails, with 3 the outcome when the 1th coin was heads, and 5 when it was tails.

We can derive the probabilities of particular outcomes by counting the number of paths that reach them. In the example, 3 has probability $\frac{1}{2}$, while 4 has probability $\frac{1}{4}$, and 5 has probability $\frac{1}{4}$. Importantly, intensional distributions have enough structure to represent correlations: We can see that we always get a 3 when the 1th coin flip is heads, regardless of whether the 0th coin flip was heads or tails. Conversely, the distribution ‹‹3 3› ‹4 5›› ascribes outcomes 3, 4, and 5 the same probabilities as ‹‹3 4› ‹3 5››, but represents the situation in which the we always get 3 when 0th coin flip is heads. An equivalent representation of ‹‹3 3› ‹4 5›› is ‹3 ‹4 5›››. Although the 3 only appears once, it is logically extended to the larger sub-tree ‹3 3› for the purposes of counting. To compute a probability, all paths are considered of a fixed length equal to the height of the tree, and shorter sub-trees are extended to copy leaves that appear at shorter height. Trees are equal $=$ when they are syntactically equal modulo these extensions.

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

probability works as usual.

Finally, looking at the middle right of the figure, consider the monadic operations used by the semantics in Figure 4.9. The $\mathrm{bit}(N)$ operation produces a uniform distribution of bits following the Nth coin flip, where the outcomes are entirely determined by the Nth flip, i.e., independent of the flips that preceded it, which appear higher in the tree. $\mathrm{return}(a)$ simply returns a—this corresponds to a point distribution of a since it is the outcome in all possible worlds (recall $a[p] = a$ for all p). Lastly, $\mathrm{bind}(\hat{x}, f)$ applies f to each possible world in $\hat{x}$, gathering up the results in an intensional distribution tree that is of equal or greater height to that of $\hat{x}$; the height could grow if f returns a tree larger than $\hat{x}$, and $\mathrm{bind}(\hat{x}, f)[p] = f(\hat{x}[p])[p]$ for all paths p.

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

$$\langle(\underline{\sigma}, \langle \texttt{bitv}_P(\texttt{I}), \texttt{cast}_{\mathrm{P}}(\texttt{flipv}(\langle \texttt{I}\ \texttt{O}\rangle))\rangle)\ (\underline{\sigma}, \langle \texttt{bitv}_P(\texttt{O}), \texttt{cast}_{\mathrm{P}}(\texttt{flipv}(\langle \texttt{I}\ \texttt{O}\rangle))\rangle)\rangle$$

In this distribution of configurations there are two worlds—the left configuration occurs when the 0th coin flip is heads, and right when it is tails. Inside of each of these configurations is a distributional value $\texttt{flipv}(\langle \texttt{I}\ \texttt{O}\rangle)$, where once again the left side is due to the coin flip being heads, and the right side being tails. *Both are relative to the same coin flip.* As such, there are two "unreachable" paths in the inner

trees: the right-branch of the left distributional value, and the left branch of the right distributional value, shown here with bullets:

$$\langle(\underline{\sigma}, \langle \mathtt{bitv}_P(\mathtt{I}), \mathtt{cast}_\mathtt{P}(\mathtt{flipv}(\langle \mathtt{I}\ \bullet\rangle))\rangle)\ (\underline{\sigma}, \langle \mathtt{bitv}_P(\mathtt{O}), \mathtt{cast}_\mathtt{P}(\mathtt{flipv}(\langle \bullet\ \mathtt{O}\rangle))\rangle)\rangle$$

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

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

Projection is defined in Figure 4.11. The definition is a straightforward use of bind to recursively flatten embedded distributional values. In our example, the projection of the mixed term before the step shows what is left after discarding the unreachable distribution elements:

$$\hat{\lceil}\langle(\underline{\sigma}, \langle \mathtt{bitv}_P(\mathtt{I}), \mathtt{cast}_\mathtt{P}(\mathtt{flipv}(\langle \mathtt{I}\ \mathtt{O}\rangle))\rangle)\ (\underline{\sigma}, \langle \mathtt{bitv}_P(\mathtt{O}), \mathtt{cast}_\mathtt{P}(\mathtt{flipv}(\langle \mathtt{I}\ \mathtt{O}\rangle))\rangle)\rangle\hat{\rceil}$$
$$= \langle(\underline{\sigma}, \langle \mathtt{bitv}_P(\mathtt{I}), \mathtt{cast}_\mathtt{P}(\mathtt{flipv}(\mathtt{I}))\rangle)\ (\underline{\sigma}, \langle \mathtt{bitv}_P(\mathtt{O}), \mathtt{cast}_\mathtt{P}(\mathtt{flipv}(\mathtt{O}))\rangle)\rangle$$

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

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

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

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

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

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

Figure 4.11: Mixed Semantics Projection

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

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

Lemma 4.3.0.1 (Simulation (Mixed))**.** *If* e *is a source expression, then* $\lceil \underline{\text{nstep}}(N, \varnothing, e) \rceil = \text{nstep}_{\mathcal{I}}(N, \varnothing, e)$.

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

Lemma 4.3.0.2 (Simulation (Intensional))**.** $\Pr[\text{nstep}_{\mathcal{I}}(N, \varnothing, e) \doteq t] = \Pr[\text{nstep}_{\mathcal{D}}(N, \varnothing, e) \doteq t]$.

4.3.5 Mixed Semantics Typing

Our type system aims to ensure that $\texttt{cast}_P$ will produce `I` and `O` with equal probability, meaning neither outcome leaks information. We establish this invariant in the PMTO proof as a consequence of type preservation for mixed terms. The mixed term typing judgment extends typing of source-program expressions (Figure 4.6) with some additional elements, and considers non-source values.

The judgment has the form $\Psi, \Phi, \Sigma \vdash \underline{\varsigma} : \tau, \Psi$, and is shown at the bottom of Figure 4.12. Here, Σ is a *store context*, which maps store locations to types; it is used to type the store $\underline{\sigma}$ in rules Store-Cons and LocV as usual. Φ represents *trace history* which encodes the exact sequence of evaluation steps taken to reach the present one. The type system reasons about the probability of distributional values conditioned on this trace history having occurred. The Ψ is an *fbset*, which is a technical device used to collect all distributional bit values $\hat{b}$ that appear in $\underline{\varsigma}$. Per the top of the figure, the fbset is a pair (Ψ^F, Ψ^B), where Ψ^F is a *flipset* containing those $\hat{b}$ that appear inside of flip values, and Ψ^B is a *bitset* containing those $\hat{b}$ inside bit values. The latter is a map from a region ρ to a set of bit values in that region. The Ψ to the right of the

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

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

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

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

$\boxed{\Psi, \Phi \vdash \hat{b} : \mathtt{flip}^\rho}$

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

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

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

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

$\ldots$

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

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

$\ldots$

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

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

$\boxed{\Psi, \Phi, \Sigma \vdash \underline{\sigma} \,;\, \Psi}$

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

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

Figure 4.12: Mixed Semantics Typing

turnstile contains all of the flip and secret bit values in the configuration itself, while the Ψ to the left of it captures those in the evaluation context and store.

The expression typing judgment $\Psi, \Phi, \Sigma, \Gamma \vdash \underline{e} : \tau \ ; \ \Gamma, \Psi$ is similar but includes variable contexts Γ as in the source-program type rules. We can see secret bit values being added to Ψ^B in the BitV-S rule, where Ψ^B is the singleton map from ρ, the region of the bit value, to $\{\hat{b}\}$, while Ψ^F is empty. Conversely, in the FlipV rule Ψ^B is empty while Ψ^F is the singleton set $\{\hat{b}\}$. We can see the maintenance of Ψ to the left of the turnstile in the Tup rule. Recursively typing the pair's left component $\underline{e}_1$ yields fbset Ψ_1 to the right of the turnstile, which is used when typing $\underline{e}_2$, and vice versa; the Store-Cons rule similarly handles the store and the expression. The rules combine two fbsets using the $\uplus$ operator. Per the top of the figure, it acts as disjoint union for flipsets but normal union for bitsets, mirroring the handling of affine and universal variables.

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

We prove a type preservation lemma to establish that these invariants are preserved.

Lemma 4.3.0.3 (Type Preservation)**.** *If* e *is a closed source expression,* $\underline{t}{\cdot}\underline{\varsigma} \in \mathrm{support}(\underline{\mathrm{nstep}}(N, \varnothing, e))$ *and* $\vdash e : \tau$*, then there exists* Σ *and* Ψ *s.t.* $\Phi, \Sigma \vdash \underline{\varsigma} : \tau, \Psi$ *where* $\Phi \triangleq [\underline{\mathrm{nstep}}(N, \varnothing, e) \doteq \underline{t}{\cdot}\underline{\varsigma}]$.

When a configuration takes any number of steps, the resulting configuration is

well-typed under new trace history Φ. Updating Φ is not arbitrary—it is necessary to satisfy a proof obligation as used in a later lemma (PMTO (Mixed)). The new Σ and Ψ are new store typings (in case new references were allocated), and the new fbset (in case flip values were either created or consumed). The proof of preservation uses a sublemma which shows typesafe substitution; this lemma makes crucial use of affinity to ensure that aggregated $\Psi_1 \uplus \Psi_2$ in contexts for compound expressions (e.g., pairs) are truly disjoint, which will be true only because the substitution is guaranteed to only occur in Ψ_1, Ψ_2, or neither, but not both.

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

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

4.3.6 Proving PMTO

To prove PMTO (Proposition 4.3.1) we first prove a variant of it for the mixed semantics, and then apply a few more lemmas to show that PMTO holds for the standard semantics too.

Lemma 4.3.0.4 (PMTO (Mixed))**.** *If* $\underline{e}_1$ *and* $\underline{e}_2$ *are closed source expressions,* $\vdash \underline{e}_1 : \tau$, $\vdash \underline{e}_2 : \tau$ *and* $\underline{e}_1 \sim \underline{e}_2$, *then (1)* $\underline{\text{nstep}}(N, \varnothing, \underline{e}_1)$ *and* $\underline{\text{nstep}}(N, \varnothing, \underline{e}_2)$ *are*

defined, and (2) $\underline{\text{nstep}}(N, \varnothing, \underline{e}_1) \approx_\sim \underline{\text{nstep}}(N, \varnothing, \underline{e}_2)$.

The judgment $\underline{e}_1 \sim \underline{e}_2$ in the premise indicates that the two expressions are *low equivalent*, meaning that the adversary cannot tell them apart. The definition of this judgment is basically standard (given in the Appendix) and we can easily prove that it is implied by $\text{obs}(e_1) = \text{obs}(e_2)$ for source expressions. Mixed PMTO establishes equivalence of the distributions of mixed configurations modulo low-equivalence. We define two distributions as equivalent modulo an underlying equivalence relation as follows:

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

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

We prove PMTO (Mixed) by induction over steps N and then unfolding the monadic definition of $\underline{\text{nstep}}(N+1)$. The induction appeals to a single-step PMTO sublemma. (As mentioned in Section 4.3.3, such a proof would not have been possible in the standard semantics.) To use this one-step PMTO sublemma, it must be that the configuration at N steps is well-typed w.r.t. current trace history Φ; we get this well-typing w.r.t. Φ from Type Preservation, discussed earlier.

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

Lemma 4.3.0.5 (Low-equivalence Soundness)**.** *If* $\hat{\underline{t}_1} \approx_{\sim} \hat{\underline{t}_2}$ *then* $\widehat{\mathrm{obs}}(\hat{\lceil}\hat{\underline{t}_1}\hat{\rceil}) \approx_{=} \widehat{\mathrm{obs}}(\hat{\lceil}\hat{\underline{t}_2}\hat{\rceil})$.

In this lemma we use a lifting of obs for intensional distributions, written $\widehat{\mathrm{obs}}$; its definition is identical to $\widetilde{\mathrm{obs}}$ in Figure 4.7 but with the intensional distribution monad $\mathcal{I}$ instead of $\mathcal{D}$.

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

Theorem 4.3.1 (PMTO)**.**

If: e_1 *and* e_2 *are closed source expressions,* $\vdash e_1 : \tau$, $\vdash e_2 : \tau$ *and* $\mathrm{obs}(e_1) = \mathrm{obs}(e_2)$

Then: (1) $\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_1)$ *and* $\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_2)$ *are defined*

And: (2) $\widetilde{\mathrm{obs}}(\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_1)) = \widetilde{\mathrm{obs}}(\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_2))$.

Proof.

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

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

□

A detailed proof is given in the Appendix C.2.1.

4.4 Implementation and Tree-based ORAM Case Study

We have implemented an interpreter and type checker for a language that extends $\lambda_{\mathbf{Obliv}}$ in several (straightforward) ways. First, we add natural number literals and random values; these can be encoded in $\lambda_{\mathbf{Obliv}}$ as fixed-width tuples of `bitv` and `flipv` respectively. We write them annotated with a security level, e.g., `2 S` or `2 P`, and write **`rnd`** `R ()` to generate a random number at region `R`. We write `natS` to be the type of a secret number in region $\bot$; `natP` for the type of a public number; `R natS` for the type of a secret number in the region `R`. We also write `R` **`rnd`** to be the type of a random

natural number in the region R. Second, we add arrays; in our code examples, we write a[n] and a[n] ← e to read and write array elements. An array of length N can be encoded in $\lambda_{\textbf{Obliv}}$ as an N-tuple of references, using nested conditional expressions to access the correct (public) index and swapping out affine contents, as must be done with references. Finally, we add records, which are like tuples but permit field accessor notation, r.x; if x is affine, doing so only consumes the field x rather than consuming all of r.

To demonstrate the expressiveness of $\lambda_{\textbf{Obliv}}$, we have used our extended language to program (and type check) a series of interesting oblivious algorithms. Section 4.4.2 presents a modern *non-recursive, tree-based ORAM* (NORAM), which is a key component of state-of-the-art ORAM implementations [160, 169, 175]. To our knowledge, ours is the first implementation automatically verified to be oblivious. Building on this NORAM, Section 4.4.3 presents a full *recursive* ORAM. Type checking it requires some advanced (but standard) language features we have not implemented, including region polymorphism, recursive and variant types, and existential quantification. Finally, the appendix presents a mostly complete implementation of *oblivious stacks* (ostacks), a kind of oblivious data structure [174] that builds on top of NORAM. The $\lambda_{\textbf{Obliv}}$ type system is not powerful enough to reason that ostacks' use of NORAM is safe; the region ordering requirement is too strong. Sections 5.8 and **??** discuss integrating $\lambda_{\textbf{Obliv}}$'s type system with a general-purpose logic as a way to potentially overcome this limitation. Our type checker and all the examples are online at `https://github.com/plum-umd/oblivml`.

4.4.1 Tree-based ORAM: Overview

A complete ORAM implements the same API as a standard array: A read operation takes an ORAM oram and index i as arguments, and returns data d stored at that index; a write operation updates oram at i with a given d. We assume that the ORAM

contents and the indexes are not visible to the adversary (i.e., they are encrypted). A simple implementation is a *Trivial ORAM*. It consists of an array of N "buckets," each of which consists of an index i and data d. A read at index j iterates over the entire array and retrieves the data associated with j, if present. The data is returned when the iteration is complete (or a default value is returned, if j is not present). Since each read touches every bucket, nothing is leaked about i. Of course, this is very inefficient—the read takes time $O(N)$ where N is the size of the array. (The code example in Figure 4.1(b) does something similar.)

A tree-based ORAM [160, 169, 175] offers better performance. It breaks its implementation into two parts. The first is a tree-like structure noram for storing the actual data blocks; this is called a *non-recursive ORAM* (or NORAM) for reasons that will be clear in the next subsection. The second part is the *position map* pm that maps logical data block indexes to *position tags* that indicate the block's position in the tree.

NORAMs do not implement read and write operations directly; instead they implement two more-primitive operations called noram_readAndRemove (or noram_rr, for short) and noram_add. The former reads the designated data block from noram and also removes it, while the latter adds the given data. Putting it all together, a Tree ORAM read from index i works in four steps: (1) retrieve tag t from pm[i]; (2) call noram_rr noram i t to remove the data d at i using t to assist the lookup; (3) update pm[i] with a randomly generated tag t2; and (4) call noram_add noram i t2 d to add back data d, but with the new tag, before returning it. An ORAM write has the same four steps, but in step (4) we add the provided data, rather than the original. (A fifth step in both cases, *eviction*, will be explained later.) As with the example in Figure 4.1(c), non-recursive ORAM combines randomness (and its tree structure) to avoid having $O(N)$ cost for the entire map: Under the right assumptions, these operations take time $O(\log(N))$.

The position tags mask the relationship between a logical index and the location of its corresponding data block in the tree. As blocks are read and written, they are shuffled around in the tree, and their new locations are recorded in the position map. As such, two ORAM read operations to the same index i will involve different access patterns in a way that leaks nothing about the index *assuming* lookups and updates to the position map itself leak no information. This assumption could be satisfied by making the position map a Trivial ORAM, but then we would lose our performance benefits. In the next subsection we simply assume we have a leak-free position map and in Section 4.4.3 we show how one can be obtained by efficiently storing the position map *recursively* in the NORAM tree structure itself.

4.4.2 Tree-based Non-recursive ORAM

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

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

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

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

The region `R ∨ R'` should be read as "`R` join `R`'" and corresponds to the join operation, $\sqcup$, over regions ρ in Section 5.2. Notice that we have `R ⊏ R ∨ R'`, which will be important when discussing well-typedness of **mux** in the discussion that follows. We choose type `(R ∨ R' rnd) * (R ∨ R' rnd)` for the data portion to illustrate that affine values can be stored in the NORAM, and to set up our implementation of full, recursive ORAM, next.

Operations The code for noram_rr is given below; we explain it just afterward.

```
1  let rec trivial_rr_h (troram : bucket) (idx : R natS) (i : natP) (acc : block) : block =
2    if i = length(troram) then acc
3    else
4      (* read out the current block, replace with dummy *)
5      let curr = bucket[i] ← (dummy_block ()) in
6      (* check if the current block is non-dummy, and its index matches the queried one *)
7      let swap : R bitS = !curr.is_dummy && curr.idx = idx in
8      let (curr, acc) = mux(swap, acc, curr) in
9      (* when swap is false , this equivalent to writing the data back; otherwise, acc
10        stores the found block and is passed into the next iteration *)
11     let _ = bucket[i] ← curr in
12     trivial_rr_h troram idx (i + 1) acc
13
14 let trivial_rr (troram : bucket) (idx : R natS) : (R ∨ R' rnd) * (R ∨ R' rnd) =
15   let ret : block = trivial_rr_h troram idx 0 (dummy_block ()) in
16   ret.data
17
18 let rec noram_rr_h (noram : noram) (idx : R natS) (tag : natP) (level : natP) (acc : block)
       : block =
19   (* compute the first index into the bucket array at depth level *)
20   let base : natP = (pow 2 level) - 1 in
21   if base >= length(noram) then acc
22   else
23     let bucket_loc : natP = base + (tag & base) in (* the bucket on the path to access *)
24     let bucket = noram[bucket_loc] in
25     let acc = trivial_rr_h bucket idx 0 acc in
26     noram_rr_h noram idx tag (level + 1) acc
27
28 let noram_rr (noram : noram) (idx : R natS) (tag : natP) : (R ∨ R' rnd) * (R ∨ R' rnd) =
29   let ret = noram_rr_h noram idx tag 0 (dummy_block ()) in
30   ret.data
```

noram_rr takes the NORAM noram and the index idx of the desired element as arguments. The tag argument is the position tag, which identifies a path through the noram binary tree along which the indexed value will be stored, if present. This tag's type natP means it is publicly visible. Initially it is stored, secretly, in the position map, but prior to passing it to this function it must be revealed (via **castP**) because it (or derivatives of it) will be used to index the arrays that make up the NORAM,

and array indexes are always adversary-visible.

noram_rr works by calling noram_rr_h which recursively works its way down the identified path. It maintains an accumulator, acc : block, over the course of the traversal. Initially, acc is a dummy block. The dummy_block () is a function call rather than a constant because the block record contains `data: (R ∨ R' rnd) * (R ∨ R' rnd)`. This member of the record must be generated fresh for each new block, since its contents are treated affinely. Each recursive call to noram_rr_h moves to a node the next level down in the tree, as determined by the tag. At each node, it reads out the bucket array, which as mentioned earlier is essentially a Trivial ORAM. The trivial_rr function calls trivial_rr_h to iterate through the entire bucket, to obliviously read out the desired block, if present.

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

This algorithm for noram_rr will access $\log N$ buckets (where N is the number of buckets in the noram), and each bucket access causes a trivial_rr which takes time b where b is the size of each bucket. Therefore, the noram_rr operation above takes time $O(b \log N)$. In the state-of-the-art ORAM constructions, such as Circuit ORAM [175], b can be parameterized as a constant (e.g., 4), which renders the overall time complexity of noram_rr to be $O(\log N)$. This is asymptotically faster than implementing the entire ORAM as a Trivial ORAM, which takes time $O(N)$.

The noram_add routine has the following signature:

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

Like the noram_rr operation, it takes an index and a position tag, but here the position

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

To avoid overflowing the root's bucket due to repeated noram_adds, our NORAM employs an additional eviction routine. It is called after both noram_add and noram_rr, to move blocks closer to the leaf buckets. This routine maintains the key invariant that each data block should reside on the path from the root to the leaf corresponding to its position tag. Different tree-based ORAM implementations differ only in their choices of b and the eviction strategies. The simple eviction strategy we implement (due to Shi et al. [160]) 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.

4.4.3 Recursive ORAM

As described in Section 4.4.1, a complete ORAM combines a non-recursive ORAM with a position map. So far, we have not said where the position map should be stored, and how. One approach is to implement it as just a regular array stored in hidden memory, e.g., on-chip (invisible to the adversary) in a secure processor deployment of ORAM (see Section 4.1.1). However, this is not possible for MPC-based deployments, in which both parties secret-share the map, and thus the adversary can observe the access pattern on the map itself. To block this side channel, we could implement the position map itself as an ORAM, e.g., a Trivial ORAM. But to do so would ruin the efficiency gain of our tree-based NORAM, since the position map lookup would have time $O(N)$, as compared to $O(\log(N))$ time for noram_rr and noram_add.

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

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

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

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

We implement the tree_rr as a call to the function tree_rr_h, which takes an additional public level argument, to indicate at which point in the list of orams to start its work (initially, 0).

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

```
9       let _ = tree_add_h oram (idx / 2) (level + 1) (r0', r1') in
10      noram_rr norams[level] idx (castP tag)
11
12  let tree_rr (oram : oram) (idx : natS): (R ∨ R' rnd) ∗ (R ∨ R' rnd) =
13    tree_rr_h oram idx 0
```

In the code above, the level indicates the embedded NORAM from which to read. For example, when level is 0, the data NORAM should be read. For any other level > 0, the NORAM will be one of the embedded position maps. Recall that each NORAM at level i has its position map at level $i + 1$, with the exception of the very last NORAM which uses a Trivial ORAM for its position map. The recursive call to tree_rr_h on line 6 reads out of the next level's map, returning the pair (r0, r1). These are the two possible position tags for nrorams[level]—we should return r0 if idx % 2 = 0 and r1 if idx % 2 = 1. The muxes on lines 7 and 8 obliviously achieve this, reading the proper result into tag, replacing it with a freshly generated tag, to satisfy the affinity requirement. Line 9 writes the updated block (r0', r1') for idx / 2 back, using an analogous tree_add_h routine, for which a level can be specified. Finally, line 10 reveals the retrieved position tag for index idx, so that it can be passed to noram_rr. Since level 0 corresponds to the actual data of the ORAM, that is what will finally be returned to the client.

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

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

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

level of the noram.

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

Discussion Unfortunately (as astute readers may have noticed), the code snippet for add will not type check. In particular, the sec_tag argument has type `R ∨ R' natS` but noram_add requires it to have type R natS. This is because the position tags for the noram at level are stored as the data of the noram at level + 1, and these are in different regions. We cannot put them in the same region because we require a single noram's metadata to have a strictly smaller region than its data (i.e., R ⊏ R ∨ R').

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

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

We re-present the definitions for the elements of noram, which we now parameterize with polymorphic region variables. For block, we add the constraint that R1 ⊏ R2 . When originally presenting NORAM, this wasn't needed because we were using concrete regions—notice that R and R ∨ R' from our previous noram definition satisfy the constraint on R1 and R2, respectively, in the new definition. Type oram is also parameterized by region variables, and is now a recursive variant: it can be either a trivial ORAM or a recursive ORAM. The latter is an NORAM paired with an ORAM, which acts as its position map. Importantly, the region R2 of the ORAM data is properly ordered with the region of the position map R1. The code would

be roughly the same as the code given above, except that rather than indexing the norams array at each recursive level, it simply recurses down the oram datastructure. Constructing such a datastructure would require satisfying the region constraints at each level, which is easy to do by simply using distinct regions for each region variable. Along with our other code examples at `https://github.com/plum-umd/oblivml`, we show how this could work using OCaml-style functors.

4.5 Oblivious Data Structures

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

4.5.1 Tree ORAM-based Oblivious Data Structures

While using a full oram can work, it is space-inefficient: an oram of size n requires a position map of size n, even if the stack contains only a few elements. Wang et al. [174] proposed a clever way to reduce this overhead: Use a tree_oram, but replace the full ORAM's complete (size n) position map with one based on the data structure's API. We will generically refer to Wang et al.'s construction as an *oblivious data structure* (ODS). For oblivious stacks, we have:

```
1  type cldata  = rnd * string<S>
2  type ostack = key ref  * rnd ref  * tree_oram
3
4  val  empty : nat → nat → ostack
5  let  empty n m = (ref 0, ref  rnd, tree_create  n  m)
6
```

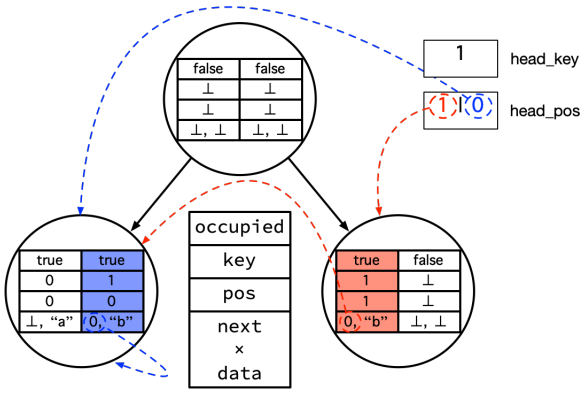


Figure 4.13: Visualizing an OStack after a push of "a" and then two possible outcomes (either blue or red) of a push of "b".

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

An oblivious stack is a triple of a key, a (**rnd**) position tag, and a Tree ORAM. The first two components form a size-1 position map which points to the head of the stack (the only element a client can access via the stack API); the head's key corresponds to the length of the stack (so it starts as 0). The position maps of the non-head stack elements are stored in the stack itself. In particular, type cldata contains the client's data in its second component, and the **rnd** component of the next element's position map in the first; the key is the current element's key, minus one. stackop takes a stack, a secret boolean indicating either push or pop (ispush), and some client data. If the operation is a push (ispush = true), stackop creates a new cldata object containing the

pushed data and the current head's **rnd** tag ((hp,v) on line 15). It then calls tree_write with a fresh position tag and new key to add the new object to the tree_oram; the new key is the old head's key plus one. Finally, it updates the current head to contain the new key and tag. If the operation is a pop, stackop looks up the head and returns the client data but also the pointer to the next element in the stack (next on line 20), which becomes the new head. The implementation of stackop ignores the overflow bit returned by tree_write. Doing so matches the behavior described by Wang et al. [174], which (we assume) aims to make an overflow adversary-*in*visible, thereby preserving PMTO. As we show in this section, ignoring overflow actually does the opposite, i.e., it compromises PMTO. The stackop code uses an **if** expression for clarity. Since the ispush variable is considered secret, a real implementation would need to **mux** instead. **??** shows how to implement ostack using **mux**.

Figure 4.13 shows the configuration of an ODS stack after two pushes. The pair head_key,head_pos are the pointer to the head of the stack (we depict the position tag as either 1 or 0 since the figure considers two possible executions for the second push; see below). Each block in the Tree ORAM has the usual fields: the occupied bit occupied, the key key, position tag pos, and client data cldata. The first push generates a fresh position tag, which happens to be 0. We add the block (true, head_key, 0, (head_pos, "a")) = (true, 0, 0, (⊥, "a")) to the Tree ORAM,[4] and it is evicted left because its tag is 0. The head_key is incremented, and head_pos is updated to 0. An identical procedure describes the second push, but in Figure 4.13 we instead show both possible outcomes for the fresh, random position tag, p. Blue indicates the outcome p = 0 and red indicates p = 1. We add the block (true, head_key + 1, p, (head_pos, "b")) = (true, 1, p, (0, "b")). The dashed arrows in Figure 4.13 indicate the bucket to which the associated key and position tag refer, revealing the abstract linked-list structure.

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

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

Figure 4.14: Distribution of ρ conditioned on $\gamma = 0$.

4.5.2 Tree ORAM-based Stack is not PMTO

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

To see how, consider the blue configuration in Figure 4.13. This Tree ORAM configuration results from pushing `"a"` and `"b"` onto the stack with position tags α and β respectively, with $\alpha = \beta = 0$. The Trivial ORAM associated with the left child is full. Consider the unlucky situation in which the value `"c"` is pushed onto the stack with a generated position tag, γ, of `0`. The `head_key` and `head_pos` are updated to `2` and `0` respectively but the block containing `"c"` is not added to the underlying Tree ORAM due to overflow. If a pop operation is executed, γ is revealed to the adversary and the position tag, ρ, will be returned to the client. Under most executions, the client will be returned `"c"` and the returned position tag will be $\rho = \beta$. However, in the overflowing execution, the client will instead receive garbage. The returned position tag is $\rho = \delta$ where δ is some fresh, uniform position tag.

Figure 4.14 shows the distribution of ρ conditioned on the observation that $\gamma = 0$.

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

4.6 Related Work

Lampson first pointed out various covert, or "side," channels of information leakage during a program's execution [104]. 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 [113, 96], power consumption [96], or memory-traces [114, 111, 147, 62]; program analysis techniques that formally ensure that a program has bounded or no leakage through instruction traces [121], timing channels [121, 3, 184, 149, 186], or memory traces [**csf13**, 109, 111]; 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 [6, 17, 183], and on preventing memory-trace leakage [73, 71, 160, 169, 175, 174, 180, 30, 76, 58, 40]. 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 [111, 147, 62, 184, 186].

Our work belongs to a large category of work that aims to statically enforce *noninterference*, e.g., by typing [172, 150]. **csf13**, Liu et al., Liu et al. developed a type system that ensures programs are MTO, generalizing a line of prior works on (language-enforced) timing channel security [3], program counter security [121]. In Liu et al's work, types are extended to indicate where values are allocated; as per our above example data can be public or secret, but can also reside in ORAM. Trace events are extended to model ORAM accesses as opaque to the adversary (similar to the Dolev-Yao modeling of encrypted messages Dolev and Yao [56]): the adversary knows that an access occurred, but not the address or whether it was a read or a write. Liu et al's type system enforces obliviousness of deterministic programs that use (assumed-to-be-correct) ORAM. $\lambda_{\textbf{Obliv}}$'s key advance is that it applies to *probabilistic* programs. It need not assume the existence of ORAM as a primitive; rather, $\lambda_{\textbf{Obliv}}$'s probabilistic nature is sufficient to allow us to *program* ORAM, per Section 4.4. Thus we can express state-of-the-art algorithmic results and formally reason about the security of their implementations, building a bridge between algorithmic and programming language techniques.

ObliVM [112] is a language for programming probabilistically oblivious algorithms intended to be run as secure multiparty computations [179]. Its type system also employs affine types to ensure random numbers are used at most once. However, it provides no mechanism to disallow constructing a non-uniformly distributed random number. When such random numbers are generated, they can be distinguished by an attacker from uniformly distributed random numbers when being revealed. Therefore, the type system in ObliVM does not guarantee obliviousness. $\lambda_{\textbf{Obliv}}$'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 mecha-

nism (and the others in $\lambda_{\mathbf{Obliv}}$) are sufficient to prove PMTO.

Our probabilistic memory trace obliviousness property bears some resemblance to probabilistic notions of noninterference. Much prior work [151, 162, 148, 130] 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 [164, 165] 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 [101] and Mu and Clark [126]) according to entropy-based measures. Work on verifying the correctness of differentially private algorithms [18, 185, 187], essentially aims to bound possible leakage; by contrast, we enforce that *no* information leaks due to a program's execution.

Our intensional distributions—while a novel syntactic device instrumental to our proof approach—are readily interpretable as measurable sets over infinite streams of bits, and there is prior work which has considered such models such as Kozen's seminal treatment [102] among others [87, 133, 156, 140, 14]. A novelty in our model is support for conditional probabilistic reasoning. This reasoning is enabled by our interpretation of monadic bind as conditioning on outcomes, and performing sampling of new bits via operations external to monad operations; doing so is in contrast to prior work which interprets monadic bind directly as (effectively) sampling new random bits.

There is a rich history for *reasoning* about probabilistic programs [154], in particular relational properties [86, 19, 21] and program logics [23, 141], including trace properties [161], privacy properties [20, 146, 64], obliviousness properties [132], and

uniformity and independence [22]. Much of this work is focused on verification techniques for some program of interest, and not on proof techniques for establishing metatheoric properties of entire languages (e.g., via a type system).

Perhaps the most closely related program logic to our setting is Probabilistic Separation Logic (PSL) [16]. PSL is a variant of separation logic in which separating conjunction models probabilistic independence. It supports reasoning about (conditional) independence and uniformity, which are both also key ideas in $\lambda_{\mathbf{Obliv}}$. There is a similar connection between some of PSL's proof rules and $\lambda_{\mathbf{Obliv}}$'s type rules; e.g., $\lambda_{\mathbf{Obliv}}$'s Mux-Flip rule and PSL's RCond rule both reason about conditional independence. It would be interesting to explore how to embed $\lambda_{\mathbf{Obliv}}$'s type system in PSL's logic, which might simplify reasoning about security for PSL, and open up reasoning about correctness for $\lambda_{\mathbf{Obliv}}$ programs. It might also permit proofs of uniformity that $\lambda_{\mathbf{Obliv}}$'s strict region ordering currently forbid. How to combine these two is not obvious, though, as PSL works on an imperative "while" language with a fixed set of (global) variables, while $\lambda_{\mathbf{Obliv}}$ is functional, and supports dynamically-sized data structures. Interesting future work!

Chapter 5

Refining Probabilistic Bounds on Information Leakage

This chapter presents our approach for ensuring a better balance of precision and performance for dynamic enforcement of knowledge-based security policies [115]. Our approach augments PP with two new techniques which can be used to improve their precision. Both techniques begin by analyzing a program, f, using intervals as the underlying numeric abstract domain for the PP.

Our first technique is to use *sampling* to augment the result. We execute the program using the possible secret values i sampled from the posterior δ' derived from a particular output o_i. If the analysis were perfectly accurate, executing $f(i)$ would produce o_i. But since intervals are overapproximate, sometimes it will not. With many sampled outcomes, we can construct a Beta distribution to estimate the size of the support of the posterior, up to some level of confidence. We can use this estimate to boost the lower bound of the abstraction, and thus improve the precision of the estimated vulnerability.

Our second technique is of a similar flavor, but uses symbolic reasoning to magnify the impact of a successful sample. We once again execute a sample which is

consistent with the posterior distribution but this time we do so *concolically* [157], thus maintaining a symbolic formula (called the *path condition*) that characterizes the set of variable valuations that would cause execution to follow the observed path. We then count the number of possible solutions and use the count to boost the lower bound of the support (with 100% confidence).

We have formalized and proved our techniques are sound with respect to the dynamic enforcement approach of Mardziel et al. [115]. In addition, we have implemented and evaluated our approaches using a privacy-sensitive ship planning scenario. We find that our techniques provide similar precision to convex polyhedra while providing orders-of-magnitude better performance. We also observe that sampling and concolic execution can be combined for even greater precision. Additional experiments are required to determine whether or not these results generalize to a diverse set of programs.

5.1 Overview

To provide an overview of our approach, we will describe the application of our techniques to a scenario that involves a coalition of ships from various nations operating in a shared region. Suppose a natural disaster has impacted some islands in the region. Some number of individuals need to be evacuated from the islands, and it falls to a regional disaster response coordinator to determine how to accomplish this. While the coalition wants to collaborate to achieve these humanitarian aims, we assume that each nation also wants to protect their sensitive data—namely ship locations and capacity.

More formally, we assume the use of the data model shown in Figure 5.1, which considers a set of ships, their coalition affiliation, the evacuation capacity of the ship, and its position, given in terms of latitude and longitude.[1] We sometimes refer to

[1]We give latitude and longitude values as integer representations of *decimal degrees* fixed to four

Field	Type	Range	Private?
ShipID	Integer	1–10	No
NationID	Integer	1–20	No
Capacity	Integer	0–1000	Yes
Latitude	Integer	-900,000–900,000	Yes
Longitude	Integer	-1,800,000–1,800,000	Yes

Figure 5.1: The data model used in the evacuation scenario.

the latter two as a location L, with $L.x$ as the longitude and $L.y$ as the latitude. We will often index properties by ship ID, writing Capacity(z) for the capacity associated with ship ID z, or Location(z) for the location.

The **evacuation problem** is defined as follows

> Given a target location L and number of people to evacuate N, compute a set of nearby ships S such that $\sum_{z \in S} \text{Capacity}(z) \geq N$.

Our goal is to solve this problem in a way that minimizes the vulnerability to the coordinator of private information, i.e., the ship locations and their exact capacity. We assume that this coordinator initially has no knowledge of the positions or capabilities of the ships other than that they fall within certain expected ranges.

If all members of the coalition share all of their data with the coordinator, then a solution is easy to compute, but it affords no privacy. Figure 5.2 gives an algorithm the response coordinator can follow that does not require each member to share all of their data. Instead, it iteratively performs queries *AtLeast* and *Nearby*. These queries do not reveal precise values about ship locations or capacity, but rather admit ranges of possibilities. The algorithm works by maintaining upper and lower bounds on the capacity of each ship `i` in the array `berths`. Each ship's bounds are updated based on the results of queries about its capacity and location. These queries aim to be privacy preserving, doing a sort of binary search to narrow in on the capacity of each ship in the operating area. The procedure completes once `is_solution` determines the minimum required capacity is reached.

decimal places; e.g., 14.3579 decimal degrees is encoded as 143579.

```
(* S = #ships; N = #evacuees; L = island loc.; D = min. proximity to L *)
    let berths = Array.make S (0,1000)
    let is_solution () = sum (Array.map fst berths) ≥ N
    let mid (x,y) = (x + y) / 2
    let AtLeast(z,b) = Capacity(z) ≥ b
    let Nearby(z,l,d) = |Loc(z).x − l.x| + |Loc(z).y − l.y| ≤ d
    while true do
      for i = 0 to S do
        let ask = mid berths[i]
        let ok = AtLeast(i, ask) && Nearby(i,L,D)
        if ok then   berths[i] ← (ask, snd berths[i])
        else         berths[i] ← (fst berths[i], ask)
        if is_solution () then return berths
      done
    done
```

Figure 5.2: Algorithm to solve the evacuation problem for a single island.

5.1.1 Computing vulnerability with abstract interpretation

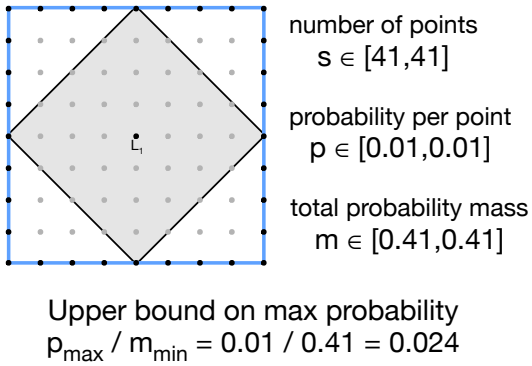
Using this procedure, what is revealed about the private variables (location and capacity)? Consider a single $Nearby(z, l, d)$ query. At the start, the coordinator is assumed to know only that z is somewhere within the operating region. If the query returns true, the coordinator now knows that s is within d units of l (using Manhattan distance). This makes $Location(z)$ more vulnerable because the adversary has less uncertainty about it.

Mardziel et al. [115] proposed a static analysis for analyzing queries such as $Nearby(z, l, d)$ to estimate the worst-case vulnerability of private data. If the worst-case vulnerability is too great, the query can be rejected. A key element of their approach is to perform abstract interpretation over the query using an abstract domain called a *probabilistic polyhedron*. An element P of this domain represents the set of possible distributions over the query's state. This state includes both the hidden secrets and the visible query results. The abstract interpretation is sound in the sense that the true distribution δ is contained in the set of distributions represented by the computed probabilistic polyhedron P.

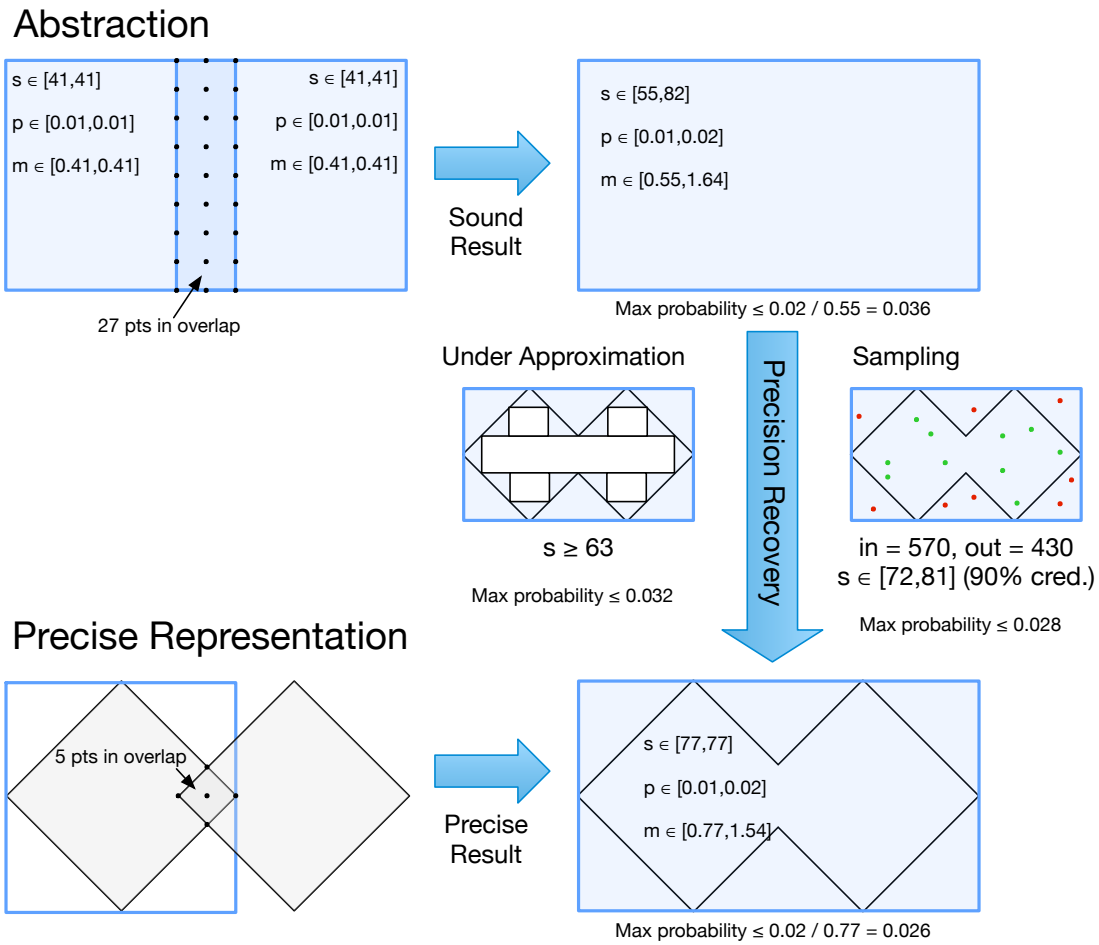
A probabilistic polyhedron P is a tuple comprising a *shape* and three *ornaments*. The shape C is an element of a standard numeric domain—e.g., intervals [46], octagons [119], or convex polyhedra [48]—which overapproximates the set of possible values in the support of the distribution. The ornaments $p \in [0, 1]$, $m \in \mathbb{R}$, and $s \in \mathbb{Z}$ are pairs which store upper and lower bounds on the probability per point, the total mass, and number of support points in the distribution, respectively. (Distributions represented by P are not necessarily normalized, so the mass m is not always 1.)

Figure 5.3(a) gives an example probabilistic polyhedron that represents the posterior of a *Nearby* query that returns true. In particular, if *Nearby(z,L_1,D)* is true then Location(z) is somewhere within the depicted diamond around L_1. Using convex polyhedra or octagons for the shape domain would permit representing this diamond exactly; using intervals would overapproximate it as the depicted 9x9 bounding box. The ornaments would be the same in any case: the size s of the support is 41 possible (x,y) points, the probability p per point is 0.01, and the total mass is 0.41, i.e., $p \cdot s$. In general, each ornament is a pair of a lower and upper bound (e.g., s_{min} and s_{max}), and m might be a more accurate estimate than $p \cdot s$. In this case shown in the figure, the bounds are tight.

Mardziel et al's procedure works by computing the posterior P for each possible query output o, and from that posterior determining the vulnerability. This is easy to do. The upper bound p_{max} of p maximizes the probability of any given point. Dividing this by the *lower bound* m_{min} of the probability mass m normalizes this probability for the worst case. For P shown in Figure 5.3(a), the bounds of p and m are tight, so the vulnerability is simply $0.01/0.41 = 0.024$.



(a) Probabilistic polyhedra



(b) Improving precision with sampling and underapproximation (concolic execution)

Figure 5.3: Computing vulnerability (max probability) using abstract interpretation

5.1.2 Improving precision with sampling and concolic execution

In Figure 5.3(a), the parameters s, p, and m are precise. However, as additional operations are performed, these quantities can accumulate imprecision. For example, suppose we are using intervals for the shape domain, and we wish to analyze the query $\mathit{Nearby}(z, L_1, 4) \vee \mathit{Nearby}(z, L_2, 4)$ (for some nearby point L_2). The result is produced by analyzing the two queries separately and then combining them with an *abstract join*; this is shown in the top row of Figure 5.3(b). Unfortunately, the result is very imprecise. The bottom row of Figure 5.3(b) illustrates the result we would get by using convex polyhedra as our shape domain. When using intervals (top row), the vulnerability is estimated as 0.036, whereas the precise answer (bottom row) is actually 0.026. Unfortunately, obtaining this precise answer is far more expensive than obtaining the imprecise one.

This paper presents two techniques that can allow us to use the less precise interval domain but then *recover* lost precision in a relatively cheap post-processing step. The effect of our techniques is shown in the middle-right of Figure 5.3(b). Both techniques aim to obtain better lower bounds for s. This allows us to update lower bounds on the probability mass m since $m_{\min}$ is at least $s_{\min} \cdot p_{\min}$ (each point has at least probability $p_{\min}$ and there are at least $s_{\min}$ of them). A larger m means a smaller vulnerability.

The first technique we explore is *sampling*, depicted to the right of the arrow in Figure 5.3(b). Sampling chooses random points and evaluates the query on them to determine whether they are in the support of the posterior distribution for a particular query result. By tracking the ratio of points that produce the expected output, we can produce an estimate of s, whose confidence increases as we include more samples. This approach is depicted in the figure, where we conclude that $s \in [72, 81]$ and $m \in [0.72, 1.62]$ with 90% confidence after taking 1000 samples, improving our vulnerability estimate to $V \leq \frac{0.02}{0.72} = 0.028$.

The second technique we explore is the use of *concolic execution* to derive a *path condition*, which is a formula over secret values that is consistent with a query result. By performing *model counting* to estimate the number of solutions to this formula, which are an underapproximation of the true size of the distribution, we can safely boost the lower bound of s. This approach is depicted to the left of the arrow in Figure 5.3(b). The depicted shapes represent discovered path condition's disjuncts, whose size sums to 63. This is a better lower bound on s and improves the vulnerability estimate to 0.032.

These techniques can be used together to further increase precision. In particular, we can first perform concolic execution, and then sample from the area not covered by this underapproximation. Importantly, Section 5.7 shows that using our techniques with the interval-based analysis yields an orders of magnitude performance improvement over using polyhedra-based analysis alone, while achieving similar levels of precision, with high confidence.

5.2 Preliminaries: Syntax and Semantics

This section presents the core language—syntax and semantics—in which we formalize our approach to computing vulnerability. We also review *probabilistic polyhedra* [115], which is the baseline analysis technique that we augment.

5.2.1 Core Language and Semantics

The programming language we use for queries is given in Figure 5.4. The language is essentially standard, apart from $\mathsf{pif}\ q\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$, which implements probabilistic choice: S_1 is executed with probability q, and S_2 with probability $1 - q$. We limit the form of expressions E so that they can be approximated by standard numeric abstract domains such as convex polyhedra [48]. Such domains require linear forms;

$$
\begin{array}{llll}
\textit{Variables} & x & \in & \mathbf{Var} \\
\textit{Integers} & n & \in & \mathbb{Z} \\
\textit{Rationals} & q & \in & \mathbb{Q} \\
\textit{States} & \sigma & \in & \mathbf{State} \stackrel{\text{def}}{=} \mathbf{Var} \rightharpoonup \mathbb{Z} \\
\textit{Distributions} & \delta & \in & \mathbf{Dist} \stackrel{\text{def}}{=} \mathbf{State} \to \mathbb{R}_{+0} \\
\textit{Arith.ops} & \textit{aop} & ::= & + \mid \times \mid - \\
\textit{Rel.ops} & \textit{relop} & ::= & \leq \mid < \mid = \mid \neq \mid \ \cdots \\
\textit{Arith.exps} & E & ::= & x \mid n \mid E_1 \ \textit{aop} \ E_2 \\
\textit{Bool.exps} & B & ::= & E_1 \ \textit{relop} \ E_2 \mid B_1 \wedge B_2 \mid B_1 \vee B_2 \mid \neg B \\
\textit{Statements} & S & ::= & \mathsf{skip} \mid x := E \mid S_1 \ ; \ S_2 \mid \mathsf{while}\ B\ \mathsf{do}\ S \mid \\
 & & & \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 \mid \mathsf{pif}\ q\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2
\end{array}
$$

Figure 5.4: Core language syntax

e.g., there is no division operator and multiplication of two variables is disallowed.[2]

We define the semantics of a program in terms of its effect on (discrete) distributions of states. States σ are partial maps from variables to integers; we write $\mathit{domain}(\sigma)$ for the set of variables over which σ is defined. Distributions δ are maps from states to nonnegative real numbers, interpreted as probabilities (in range $[0, 1]$). The denotational semantics considers a program as a relation between distributions. In particular, the semantics of statement S, written $[\![S]\!]$, is a function of the form $\mathbf{Dist} \to \mathbf{Dist}$; we write $[\![S]\!]\delta = \delta'$ to say that the semantics of S maps input distribution δ to output distribution δ'. Distributions are not necessarily normalized; we write $\|\delta\|$ as the probability mass of δ (which is between 0 and 1). We write $\dot{\sigma}$ to denote the point distribution that gives σ probability 1, and all other states 0.

The semantics is standard and not crucial in order to understand our techniques. In Appendix C.2 we provide the semantics in full. See Clarkson et al. [44] or Mardziel et al [115] for detailed explanations.

[2]Relaxing such limitations is possible—e.g., polynominal inequalities can be approximated using convex polyhedra [11]—but doing so precisely and scalably is a challenge.

5.2.2 Probabilistic polyhedra

To compute vulnerability for a program S we must compute (an approximation of) its output distribution. One way to do that would be to use sampling: Choose states σ at random from the input distribution δ, "run" the program using that input state, and collect the frequencies of output states σ' into a distribution δ'. While using sampling in this manner is simple and appealing, it could be both expensive and imprecise. In particular, depending on the size of the input and output space, it may take many samples to arrive at a proper approximation of the output distribution.

Probabilistic polyhedra [115] can address both problems. This abstract domain combines a standard domain C for representing numeric program states with additional *ornaments* that all together can safely represent S's output distribution.

Probabilistic polyhedra work for any numeric domain; in this paper we use both convex polyhedra [48] and intervals [46]. For concreteness, we present the defintion using convex polyhedra. We use the meta-variables β, β_1, β_2, etc. to denote linear inequalities.

Definition 5.2.1. *A* convex polyhedron $C = (B, V)$ *is a set of linear inequalities* $B = \{\beta_1, \ldots, \beta_m\}$*, interpreted conjunctively, over variables* V*. We write* $\mathbb{C}$ *for the set of all convex polyhedra. A polyhedron* C *represents a set of states, denoted* $\gamma_{\mathbb{C}}(C)$*, as follows, where* $\sigma \models \beta$ *indicates that the state* σ *satisfies the inequality* β*.*

$$\gamma_{\mathbb{C}}((B, V)) \stackrel{\text{def}}{=} \{\sigma \ : \ \mathit{domain}(\sigma) = V,\ \forall \beta \in B.\ \sigma \models \beta\}$$

Naturally we require that $\mathit{domain}(\{\beta_1, \ldots, \beta_n\}) \subseteq V$*; i.e.,* V *mentions all variables in the inequalities. Let* $\mathit{domain}((B, V)) = V$*.*

Probabilistic polyhedra extend this standard representation of sets of program states to sets of *distributions* over program states.

Definition 5.2.2. *A* probabilistic polyhedron *P is a tuple* $(C, \mathrm{s}^{\min}, \mathrm{s}^{\max}, \mathrm{p}^{\min}, \mathrm{p}^{\max}, \mathrm{m}^{\min}, \mathrm{m}^{\max})$. *We write $\mathbb{P}$ for the set of probabilistic polyhedra. The quantities $\mathrm{s}^{\min}$ and $\mathrm{s}^{\max}$ are lower and upper bounds on the number of support points in the concrete distribution(s) P represents. A support point of a distribution is one which has non-zero probability. The quantities $\mathrm{p}^{\min}$ and $\mathrm{p}^{\max}$ are lower and upper bounds on the probability mass per support point. The $\mathrm{m}^{\min}$ and $\mathrm{m}^{\max}$ components give bounds on the total probability mass (i.e., the sum of the probabilities of all support points). Thus P represents the* set *of distributions $\gamma_{\mathbb{P}}(P)$ defined below.*

$$\begin{aligned}\gamma_{\mathbb{P}}(P) \stackrel{\text{def}}{=} \{\delta \ : \ & \mathit{support}(\delta) \subseteq \gamma_{\mathbb{C}}(C) \wedge \\ & \mathrm{s}^{\min} \leq |\mathit{support}(\delta)| \leq \mathrm{s}^{\max} \wedge \\ & \mathrm{m}^{\min} \leq \|\delta\| \leq \mathrm{m}^{\max} \wedge \\ & \forall \sigma \in \mathit{support}(\delta).\ \mathrm{p}^{\min} \leq \delta(\sigma) \leq \mathrm{p}^{\max}\}\end{aligned}$$

We will write $\mathit{domain}(P) \stackrel{\text{def}}{=} \mathit{domain}(C)$ to denote the set of variables used in the probabilistic polyhedron.

Note the set $\gamma_{\mathbb{P}}(P)$ is a singleton exactly when $\mathrm{s}^{\min} = \mathrm{s}^{\max} = \#(C)$ and $\mathrm{p}^{\min} = \mathrm{p}^{\max}$, and $\mathrm{m}^{\min} = \mathrm{m}^{\max}$, where $\#(C)$ denotes the number of discrete points in convex polyhedron C. In such a case $\gamma_{\mathbb{P}}(P)$ contains only the uniform distribution where each state in $\gamma_{\mathbb{C}}(C)$ has probability $\mathrm{p}^{\min}$. In general, however, the concretization of a probabilistic polyhedron will have an infinite number of distributions, with per-point probabilities varied somewhere in the range $\mathrm{p}^{\min}$ and $\mathrm{p}^{\max}$. Distributions represented by a probabilistic polyhedron are not necessarily normalized. In general, there is a relationship between $\mathrm{p}^{\min}, \mathrm{s}^{\min}$, and $\mathrm{m}^{\min}$, in that $\mathrm{m}^{\min} \geq \mathrm{p}^{\min} \cdot \mathrm{s}^{\min}$ (and $\mathrm{m}^{\max} \leq \mathrm{p}^{\max} \cdot \mathrm{s}^{\max}$), and the combination of the three can yield more information than any two in isolation.

The *abstract semantics* of S is written $\langle\!\langle S \rangle\!\rangle\, P = P'$, and indicates that abstractly

interpreting S where the distribution of input states are approximated by P will produce P', which approximates the distribution of output states. Following standard abstract interpretation terminology, $\mathcal{P}(\mathbf{Dist})$ (sets of distributions) is the *concrete domain*, $\mathbb{P}$ is the *abstract domain*, and $\gamma_{\mathbb{P}} : \mathbb{P} \to \mathcal{P}(\mathbf{Dist})$ is the *concretization function* for $\mathbb{P}$. We do not present the abstract semantics here; details can be found in Mardziel et al. [115]. Importantly, this abstract semantics is sound:

Theorem 5.2.1 (Soundness)**.** *For all* $S, P_1, P_2, \delta_1, \delta_2$*, if* $\delta_1 \in \gamma_{\mathbb{P}}(P_1)$ *and* $\langle\!\langle S \rangle\!\rangle P_1 = P_2$*, then* $[\![S]\!]\delta_1 = \delta_2$ *with* $\delta_2 \in \gamma_{\mathbb{P}}(P_2)$*.*

Proof. See Theorem 6 in Mardziel et. al [115]. □

Consider the example from Section 5.1.2. We assume the adversary has no prior information about the location of ship s. So, δ_1 above is simply the uniform distribution over all possible locations. The statement S is the query issued by the adversary, $\mathit{Nearby}(z, L_1, 4) \vee \mathit{Nearby}(z, L_2, 4)$.[3] If we assume that the result of the query is `true` then the adversary learns that the location of s is within (Manhattan) distance 4 of L_1 or L_2. This posterior belief (δ_2) is represented by the overlapping diamonds on the bottom-right of Figure 5.3(b). The abstract interpretation produces a sound (interval) overapproximation (P_2) of the posterior belief. This is modeled by the rectangle which surrounds the overlapping diamonds. This rectangle is the "join" of two overlapping boxes, which each correspond to one of the *Nearby* calls in the disjuncts of S.

5.3 Computing Vulnerability: Basic procedure

The key goal of this paper is to quantify the risk to secret information of running a query over that information. This section explains the basic approach by which

[3] Appendix C.1 shows the code, which computes Manhattan distance between s and L_1 and L_2 and then sets an output variable if either distance is within four units.

we can use probabilistic polyhedra to compute *vulnerability*, i.e., the probability of the most probable point of the posterior distribution. Improvements on this basic approach are given in the next two sections.

Our convention will be to use C_1, $\mathrm{s}_1^{\min}$, $\mathrm{s}_1^{\max}$, etc. for the components associated with probabilistic polyhedron P_1. In the program S of interest, we assume that secret variables are in the set T, so input states are written σ_T, and we assume there is a single output variable r. We assume that the adversary's initial uncertainty about the possible values of the secrets T is captured by the probabilistic polyhedron P_0 (such that $\mathit{domain}(P_0) \supseteq T$).

Computing vulnerability occurs according to the following procedure.

1. Perform abstract interpretation: $\langle\!\langle S \rangle\!\rangle\, P_0 = P$
2. Given a concrete output value of interest, o, perform abstract conditioning to define $P_{r=o} \stackrel{\text{def}}{=} (P \wedge r{=}o)$.[4]

The vulnerability V is the probability of the most likely state(s). When a probabilistic polyhedron represents one or more true distributions (i.e., the probabilities all sum to 1), the most probable state's probability is bounded by $\mathrm{p}^{\max}$. However, the abstract semantics does not always normalize the probabilistic polyhedron as it computes, so we need to scale $\mathrm{p}^{\max}$ according to the total probability mass. To ensure that our estimate is on the safe side, we scale $\mathrm{p}^{\max}$ using the *minimum* probability mass: $V = \frac{\mathrm{p}^{\max}}{\mathrm{m}^{\min}}$. In Figure 5.3(b), the sound approximation in the top-right has $V \leq \frac{0.02}{0.55} = 0.036$ and the most precise approximation in the bottom-right has $V \leq \frac{0.02}{0.77} = 0.026$.

5.4 Improving precision with sampling

We can improve the precision of the basic procedure using sampling. First we introduce some notational convenience:

[4] We write $P \wedge B$ and not $P \mid B$ because P need not be normalized.

$$P_T \stackrel{\text{def}}{=} P \wedge (r = o) \downharpoonright T$$

$$P_{T+} \stackrel{\text{def}}{=} P_T \text{ revised polyhedron with confidence } \omega$$

P_T is equivalent to step 2, above, but projected onto the set of secret variables T. P_{T+} is the improved (via sampling) polyhedron.

After computing P_T with the basic procedure from the previous section we take the following additional steps:

1. Set counters α and β to zero.

2. Do the following N times (for some N, see below):

   (a) Randomly select an input state $\sigma_T \in \gamma_{\mathbb{C}}(C_T)$.

   (b) "Run" the program by computing $[\![S]\!]\dot{\sigma_T} = \delta$. If there exists $\sigma \in \mathit{support}(\delta)$ with $\sigma(r) = o$ then increment α, else increment β.

3. We can interpret α and β as the parameters of a Beta distribution of the likelihood that an arbitrary state in $\gamma_{\mathbb{C}}(C_T)$ is in the support of the true distribution. From these parameters we can compute the *credible interval* $[p_L, p_U]$ within which is contained the true likelihood, with confidence ω (where $0 \leq \omega \leq 1$). A credible interval is essentially a Bayesian analogue of a confidence interval and can be computed from the cumulative distribution function (CDF) of the Beta distribution (the 99% credible interval is the interval $[a, b]$ such that the CDF at a has value 0.005 and the CDF at b has value 0.995). In general, obtaining a higher confidence or a narrower interval will require a higher N. Let result $P_{T+} = P_T$ except that $\mathrm{s}^{\min}_{T+} = p_L \cdot \#(C_T)$ and $\mathrm{s}^{\max}_{T+} = p_U \cdot \#(C_T)$ (assuming these improve on $\mathrm{s}^{\min}_T$ and $\mathrm{s}^{\max}_T$). We can then propagate these improvements to $\mathrm{m}^{\min}$

and $\mathrm{m}^{\max}$ by defining $\mathrm{m}^{\min}_{T+} = \mathrm{p}^{\min}_{T} \cdot \mathrm{s}^{\min}_{T+}$ and $\mathrm{m}^{\max}_{T+} = \mathrm{p}^{\max}_{T} \cdot \mathrm{s}^{\max}_{T+}$. Note that if $\mathrm{m}^{\min}_{T} > \mathrm{m}^{\min}_{T+}$ we leave it unchanged, and do likewise if $\mathrm{m}^{\max}_{T} < \mathrm{m}^{\max}_{T+}$.

At this point we can compute the vulnerability as in the basic procedure, but using P_{T+} instead of P_T.

Consider the example of Section 5.1.2. In Figure 5.3(b), we draw samples from the rectangle in the top-right. This rectangle overapproximates the set of locations where s might be, given that the query returned true. We sample locations from this rectangle and run the query on each sample. The green (red) dots indicate true (false) results, which are added to α (β). After sampling $N = 1000$ locations, we have $\alpha = 570$ and $\beta = 430$. Choosing $\omega = .9$ (90%), we compute the credible interval $[0.53, 0.60]$. With $\#(C_T) = 135$, we compute $[\mathrm{s}^{\min}_{T+}, \mathrm{s}^{\max}_{T+}]$ as $[0.53 \cdot 135, 0.60 \cdot 135] = [72, 81]$.

There are several things to notice about this procedure. First, observe that in step 2b we "run" the program using the point distribution $\dot{\sigma}$ as an input; in the case that S is deterministic (has no pif statements) the output distribution will also be a point distribution. However, for programs with pif statements there are multiple possible outputs depending on which branch is taken by a pif. We consider all of these outputs so that we can confidently determine whether the input state σ could ever cause S to produce result o. If so, then σ should be considered part of P_{T+}. If not, then we can safely rule it out (i.e., it is part of the overapproximation).

Second, we only update the size parameters of P_{T+}; we make no changes to $\mathrm{p}^{\min}_{T+}$ and $\mathrm{p}^{\max}_{T+}$. This is because our sampling procedure only determines whether it is *possible* for an input state to produce the expected output. The probability that an input state produces an output state is already captured (soundly) by p_T so we do not change that. This is useful because the approximation of p_T does not degrade with the use of the interval domain in the way the approximation of the size degrades (as illustrated in Figure 5.3(b)). Using sampling is an attempt to regain the precision

lost on the size component (only).

Finally, the confidence we have that sampling has accurately assessed which input states are in the support is orthogonal to the probability of any given state. In particular, P_T is an abstraction of a distribution δ_T, which is a mathematical object. Confidence ω is a measure of how likely it is that our abstraction (or, at least, the size part of it) is accurate.

We prove (in Appendix C.2.1) that our sampling procedure is sound:

Theorem 5.4.1 (Sampling is Sound)**.**

If $\delta_0 \in \gamma_{\mathbb{P}}(P_0)$, $\langle\!\langle S \rangle\!\rangle\, P_0 = P$, *and* $[\![S]\!]\delta_0 = \delta$ *then* $\delta_T \in \gamma_{\mathbb{P}}(P_{T+})$ *with confidence* ω *where*

$$\delta_T \stackrel{\text{def}}{=} \delta \wedge (r = o) \lfloor T$$

$$P_T \stackrel{\text{def}}{=} P \wedge (r = o) \lfloor T$$

$$P_{T+} \stackrel{\text{def}}{=} P_T \textit{ sampling revised with confidence } \omega.$$

5.5 Improving precision with concolic execution

Another approach to improving the precision of a probabilistic polyhedron P is to use concolic execution. The idea here is to "magnify" the impact of a single sample to soundly increase $s^{\min}$ by considering its execution *symbolically*. More precisely, we concretely execute a program using a particular secret value, but maintain symbolic constraints about how that value is used. This is referred to as *concolic* execution [157]. We use the collected constraints to identify all points that would induce the same execution path, which we can include as part of $s^{\min}$.

We begin by defining the semantics of concolic execution, and then show how it can be used to increase $s^{\min}$ soundly.

5.5.1 (Probabilistic) Concolic Execution

Concolic execution is expressed as rewrite rules defining a judgment $\langle \Pi, S \rangle \longrightarrow^{p}_{\pi} \langle \Pi', S' \rangle$. Here, Π is pair consisting of a concrete state σ and symbolic state ζ. The latter maps variables $x \in \mathbf{Var}$ to *symbolic expressions* $\mathcal{E}$ which extend expressions E with *symbolic variables* α. This judgment indicates that under input state Π the statement S reduces to statement S' and output state Π' with probability p, with *path condition* π. The path condition is a conjunction of boolean symbolic expressions $\mathcal{B}$ (which are just boolean expressions B but altered to use symbolic expressions $\mathcal{E}$ instead of expressions E) that record which branch is taken during execution. For brevity, we omit π in a rule when it is true.

The rules for the concolic semantics are given in Figure 5.5. Most of these are standard, and deterministic (the probability annotation p is 1). Path conditions are recorded for if and while, depending on the branch taken. The semantics of $\mathsf{pif}\ q\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$ is non-deterministic: the result is that of S_1 with probability q, and S_2 with probability $1 - q$. We write $\zeta(B)$ to substitute free variables $x \in B$ with their mapped-to values $\zeta(x)$ and then simplify the result as much as possible. For example, if $\zeta(x) = \alpha$ and $\zeta(y) = 2$, then $\zeta(x > y + 3) = \alpha > 5$. The same goes for $\zeta(E)$.

We define a *complete run* of the concolic semantics with the judgment $\langle \Pi, S \rangle \Downarrow^{p}_{\pi} \Pi'$, which has two rules:

$$\langle \Pi, \mathsf{skip} \rangle \Downarrow^{1}_{\mathsf{true}} \Pi$$

$$\frac{\langle \Pi, S \rangle \longrightarrow^{p}_{\pi} \langle \Pi', S' \rangle \quad \langle \Pi', S' \rangle \Downarrow^{q}_{\pi'} \Pi''}{\langle \Pi, S \rangle \Downarrow^{p \cdot q}_{\pi \wedge \pi'} \Pi''}$$

A complete run's probability is thus the product of the probability of each individual step taken. The run's path condition is the conjunction of the conditions of each step.

$$
\begin{array}{l}
\langle(\sigma,\zeta), x \ := \ E\rangle \longrightarrow^{1} \langle(\sigma[x \mapsto \sigma(E)], \zeta[x \mapsto \zeta(E)]), \mathsf{skip}\rangle \\
\langle(\sigma,\zeta), \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \longrightarrow^{1}_{\zeta(B)} \langle(\sigma,\zeta), S_1\rangle \qquad \text{if } \sigma(B) \\
\langle(\sigma,\zeta), \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \longrightarrow^{1}_{\zeta(\neg B)} \langle(\sigma,\zeta), S_2\rangle \quad \text{if } \sigma(\neg B) \\
\langle\Pi, \mathsf{pif}\ q\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \longrightarrow^{q} \langle\Pi, S_1\rangle \\
\langle\Pi, \mathsf{pif}\ q\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \longrightarrow^{1-q} \langle\Pi, S_2\rangle \\
\langle\Pi, S_1 \ ; \ S_2\rangle \longrightarrow^{1}_{\pi} \langle\Pi', S_1' \ ; \ S_2\rangle \quad \text{if } \langle\Pi, S_1\rangle \longrightarrow^{1}_{\pi} \langle\Pi', S_1'\rangle \\
\langle\Pi, \mathsf{skip} \ ; \ S\rangle \longrightarrow^{1} \langle\Pi, S\rangle \\
\langle\Pi, \mathsf{while}\ B\ \mathsf{do}\ S\rangle \longrightarrow^{1}_{\zeta(B)} \langle\Pi, S \ ; \ \mathsf{while}\ B\ \mathsf{do}\ S\rangle \qquad \text{if } \sigma(B) \\
\langle\Pi, \mathsf{while}\ B\ \mathsf{do}\ S\rangle \longrightarrow^{1}_{\zeta(\neg B)} \langle\Pi, \mathsf{skip}\rangle \qquad \text{if } \sigma(\neg B)
\end{array}
$$

Figure 5.5: Concolic semantics

The path condition π for a complete run is a conjunction of the (symbolic) boolean guards evaluated during an execution. π can be converted to disjunctive normal form (DNF), and given the restrictions of the language the result is essentially a set of convex polyhedra over symbolic variables α.

5.5.2 Improving precision

Using concolic execution, we can improve our estimate of the size of a probabilistic polyhedron as follows:

1. Randomly select an input state $\sigma_T \in \gamma_{\mathbb{C}}(C_T)$ (recall that C_T is the polyhedron describing the possible valuations of secrets T).

2. Set $\Pi = (\sigma_T, \zeta_T)$ where ζ_T maps each variable $x \in T$ to a fresh symbolic variable α_x. Perform a complete concolic run $\langle\Pi, S\rangle \Downarrow^{p}_{\pi} (\sigma', \zeta')$. Make sure that $\sigma'(r) = o$, i.e., the expected output. If not, select a new σ_T and retry. Give up after some number of failures N. For our example shown in Figure 5.3(b), we might obtain a path condition $|Loc(z).x - L_1.x| + |Loc(z).y - L_1.y| \leq 4$ that captures the left diamond of the disjunctive query.

3. After a successful concolic run, convert path condition π to DNF, where each conjunctive clause is a polyhedron C_i. Also convert uses of disequality ($\leq$ and

$\geq$) to be strict ($<$ and $>$).

4. Let $C = C_T \sqcap (\bigsqcup_i C_i)$; that is, it is the join of each of the polyhedra in $DNF(\pi)$ "intersected" with the original constraints. This captures all of the points that could possibly lead to the observed outcome along the concolically executed path. Compute $n = \#(C)$. Let $P_{T+} = P_T$ except define $\mathrm{s}^{\min}_{T+} = n$ if $\mathrm{s}^{\min}_{T} < n$ and $\mathrm{m}^{\min}_{T+} = \mathrm{p}^{\min}_{T} \cdot n$ if $\mathrm{m}^{\min}_{T} < \mathrm{p}^{\min}_{T} \cdot n$. (Leave them as is, otherwise.) For our example, $n = 41$, the size of the left diamond. We do not update $\mathrm{s}^{\min}_{T}$ since $41 < 55$, the probabilistic polyhedron's lower bound (but see below).

Theorem 5.5.1 (Concolic Execution is Sound)**.**
If $\delta_0 \in \gamma_{\mathbb{P}}(P_0)$, $\langle\!\langle S \rangle\!\rangle\, P_0 = P$, *and* $[\![S]\!]\delta_0 = \delta$ *then* $\delta_T \in \gamma_{\mathbb{P}}(P_{T+})$ *where*

$$\delta_T \stackrel{\text{def}}{=} \delta \wedge (r = o) \downharpoonleft T$$

$$P_T \stackrel{\text{def}}{=} P \wedge (r = o) \downharpoonleft T$$

$$P_{T+} \stackrel{\text{def}}{=} P_T \textit{ concolically revised.}$$

The proof is in Appendix C.2.1.

5.5.3 Combining Sampling with Concolic Execution

Sampling can be used to further augment the results of concolic execution. The key insight is that the presence of a sound under-approximation generated by the concolic execution means that it is unnecessary to sample from the under-approximating region. Here is the algorithm:

1. Let $C = C_0 \sqcap (\bigsqcup_i C_i)$ be the under-approximating region.

2. Perform sampling per the algorithm in Section 5.4, but with two changes:

   - if a sampled state $\sigma_T \in \gamma_{\mathbb{C}}(C)$, ignore it

- When done sampling, compute $\mathrm{s}^{\min}_{T+} = p_L \cdot (\#(C_T) - \#(C)) + \#(C)$ and $\mathrm{s}^{\max}_{T+} = p_U \cdot (\#(C_T) - \#(C)) + \#(C)$. This differs from Section 5.4 in not including the count from concolic region C in the computation. This is because, since we ignored samples $\sigma_T \in \gamma_{\mathbb{C}}(C)$, the credible interval $[p_L, p_U]$ bounds the likelihood that any given point in $C_T \setminus C$ is in the support of the true distribution.

For our example, concolic execution indicated there are at least 41 points that satisfy the query. With this in hand, and using the same samples as shown in Section 5.4, we can refine $s \in [74, 80]$ and $m \in [0.74, 0.160]$ (the credible interval is formed over only those samples which satisfy the query but fall outside the under-approximation returned by concolic execution). We improve the vulnerability estimate to $V \leq \frac{0.02}{0.0.74} = 0.027$. These bounds (and vulnerability estimate) are better than those of sampling alone ($s \in [72, 81]$ with $V \leq 0.028$).

The statement of soundness and its proof can be found in Appendix C.2.1.

5.6 Implementation

We have implemented our approach as an extension of Mardziel et al. [115], which is written in OCaml. This baseline implements numeric domains C via an OCaml interface to the Parma Polyhedra Library [10]. The counting procedure $\#(C)$ is implemented by LattE [53]. Support for arbitrary precision and exact arithmetic (e.g., for manipulating $\mathrm{m}^{\min}$, $\mathrm{p}^{\min}$, etc.) is provided by the `mlgmp` OCaml interface to the GNU Multi Precision Arithmetic library. Rather than maintaining a single probabilistic polyhedron P, the implementation maintains a *powerset* of polyhedra [9], i.e., a finite disjunction. Doing so results in a more precise handling of join points in the control flow, at a somewhat higher performance cost.

We have implemented our extensions to this baseline for the case that domain

C is the interval numeric domain [46]. Of course, the theory fully applies to any numeric abstract domain. We use Gibbs sampling, which we implemented ourselves. We delegate the calculation of the beta distribution and its corresponding credible interval to the `ocephes` OCaml library, which in turns uses the GNU Scientific Library. It is straightforward to lift the various operations we have described to the powerset domain. All of our code is available at https://github.com/GaloisInc/TAMBA.

5.7 Experiments

To evaluate the benefits of our techniques, we applied them to queries based on the evacuation problem outlined in Section 5.1. We found that while the baseline technique can yield precise answers when computing vulnerability, our new techniques can achieve close to the same level of precision far more efficiently.

5.7.1 Experimental Setup

For our experiments we analyzed queries similar to $\mathit{Nearby}(s, l, d)$ from Figure 5.2. We generalize the *Nearby* query to accept a set of locations L—the query returns true if s is within d units of any one of the islands having location $l \in L$. In our experiments we fix $d = 100$. We consider the secrecy of the location of s, $\mathit{Location}(s)$. We also analyze the execution of the resource allocation algorithm of Figure 5.2 directly; we discuss this in Section 5.7.3.

We measure the time it takes to compute the *vulnerability* (i.e., the probability of the most probable point) following each query. In our experiments, we consider a single ship s and set its coordinates so that it is always in range of some island in L, so that the concrete query result returns true (i.e. $\mathit{Nearby}(s, L, 100) = \mathit{true}$). We measure the vulnerability following this query result starting from a prior belief that the coordinates of s are uniformly distributed with $0 \leq \text{Location}(s).x \leq 1000$ and

$0 \leq \text{Location}(s).y \leq 1000$.

In our experiments, we varied several experimental parameters: *analysis method* (either P, I, CE, S, or CE+S), *query complexity* c; *AI precision level* p; and *number of samples* n. We describe each in turn.

Analysis method We compared five techniques for computing vulnerability:

P: Abstract interpretation (AI) with convex polyhedra for domain C (Section 5.3),

I: AI with intervals for C (Section 5.3),

S: AI with intervals augmented with sampling (Section 5.4),

CE: AI with intervals augmented with concolic execution (Section 5.5), and

CE+S: AI with intervals augmented with both techniques (Section 5.5.3)

The first two techniques are due to Mardziel et al. [115], where the former uses convex polyhedra and the latter uses intervals (aka boxes) for the underlying polygons. In our experiments we tend to focus on P since I's precision is unacceptably poor (e.g., often vulnerability = 1).

Query complexity. We consider queries with different L; we say we are increasing the *complexity* of the query as L gets larger. Let $c = |L|$; we consider $1 \leq c \leq 5$, where larger L include the same locations as smaller ones. We set each location to be at least $2 \cdot d$ Manhattan distance units away from any other island (so diamonds like those in Figure 5.3(a) never overlap).

Precision. The precision parameter p bounds the size of the powerset abstract domain at all points during abstract interpretation. This has the effect of forcing joins when the powerset grows larger than the specified precision. As p grows larger, the results of abstract interpretation are likely to become more precise (i.e. vulnerability gets closer to the true value). We considered p values of 1, 2, 4, 8, 16, 32, and 64.

Samples taken. For the latter three analysis methods, we varied the number of samples taken n. For analysis CE, n is interpreted as the number of samples to try per polyhedron before giving up trying to find a "valid sample."[5] For analysis S, n is the number of samples, distributed proportionally across all the polyhedra in the powerset. For analysis CE+S, n is the combination of the two. We considered sample size values of $1,000 - 50,000$ in increments of $1,000$. We always compute an interval with ω =99.9% confidence (which will be wider when fewer samples are used).

System description. We ran experiments varying all possible parameters. For each run, we measured the total execution time (wall clock) in seconds to analyze the query and compute vulnerability. All experiments were carried out on a MacBook Air with OSX version 10.11.6, a 1.7GHz Intel Core i7, and 8GB of RAM. We ran a single trial for each configuration of parameters. Only wall-clock time varies across trials; informally, we observed time variations to be small.

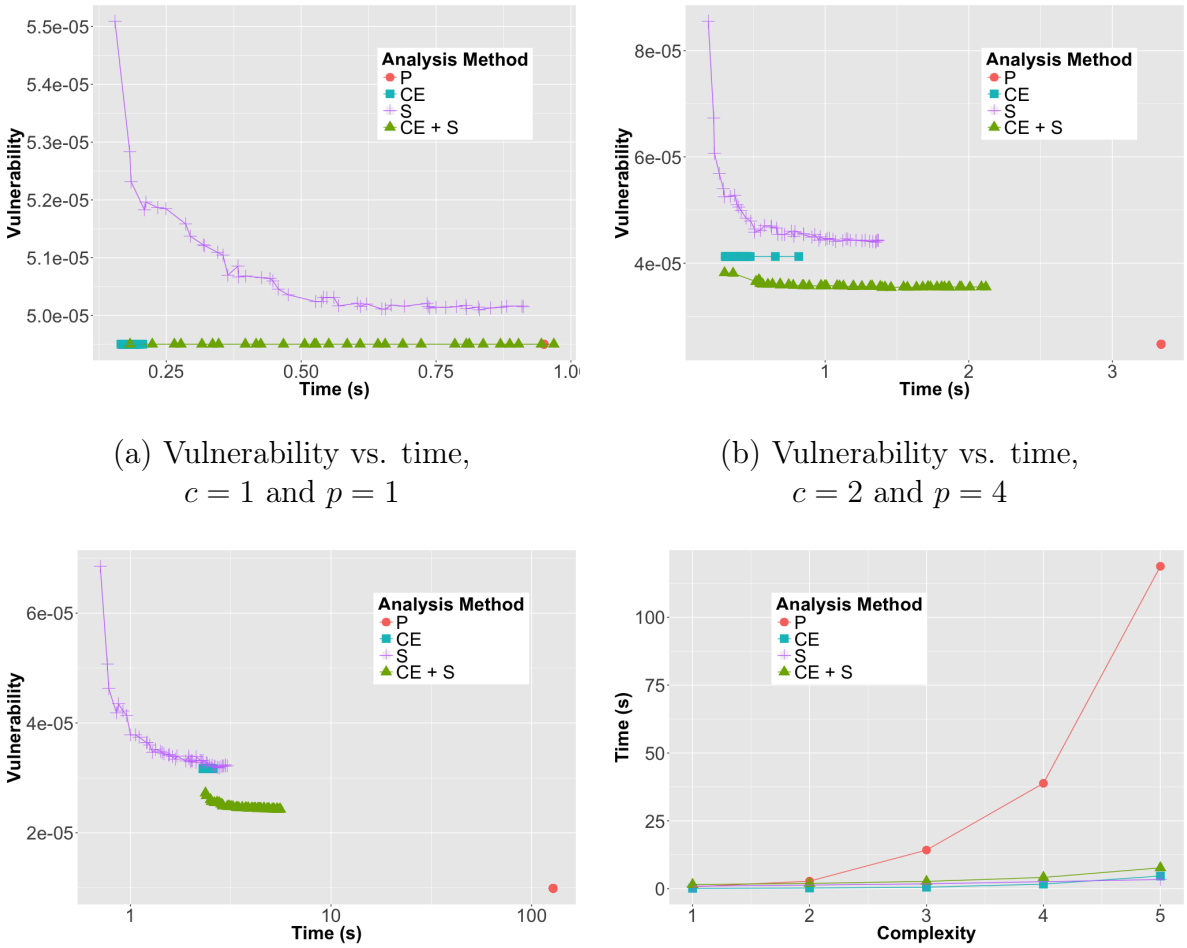
5.7.2 Results

Figure 5.6(a)–(c) measure vulnerability (y-axis) as a function of time (x-axis) for each analysis.[6] These three figures characterize three interesting "zones" in the space of complexity and precision. The results for method I are not shown in any of the figures. This is because I always produces a vulnerability of 1. The refinement methods (CE, S, and CE+S) are all over the interval domain, and should be considered as "improving" the vulnerability of I.

In Figure 5.6(a) we fix $c = 1$ and $p = 1$. In this configuration, baseline analysis P can compute the true vulnerability in $\sim$0.95 seconds. Analysis CE is also able to compute the true vulnerability, but in $\sim$0.19 seconds. Analysis S is able to compute a vulnerability to within $\sim 5 \cdot e^{-6}$ of optimal in $\sim$0.15 seconds. These data points support two key observations. First, even a very modest number of samples improves

[5] This is the N parameter from section 5.5.

[6] These are best viewed on a color display.



(a) Vulnerability vs. time, $c = 1$ and $p = 1$

(b) Vulnerability vs. time, $c = 2$ and $p = 4$

(c) Vulnerability vs. time, $c = 5$ and $p = 32$ (X-axis is log-scale)

(d) Time vs. complexity, $n = 50,000$ and $p = 64$

Figure 5.6: Experimental results

vulnerability significantly over just analyzing with intervals. Second, concolic execution is only slightly slower and can achieve the optimal vulnerability. Of course, concolic execution is not a panacea. As we will see, a feature of this configuration is that no joins take place during abstract interpretation. This is critical to the precision of the concolic execution.

In Figure 5.6(b) we fix $c = 2$ and $p = 4$. In contrast to the configuration of Figure 5.6(a), the values for c and p in this configuration are not sufficient to prevent all joins during abstract interpretaion. This has the effect of taking polygons that represent individual paths through the program and joining them into a single polygon

representing many paths. We can see that this is the case because baseline analysis P is now achieving a better vulnerability than CE. However, one pattern from the previous configuration persists: all three refinement methods (CE, S, CE+S) can achieve vulnerability within $\sim 1 \cdot e^{-5}$ of P, but in $\frac{1}{4}$ the time. In contrast to the previous configuration, analysis CE+S is now able to make a modest improvement over CE (since it does not achieve the optimal).

In Figure 5.6(c) we fix $c = 5$ and $p = 32$. This configuration magnifies the effects we saw in Figure 5.6(b). Similarly, in this configuration there are joins happening, but the query is much more complex and the analysis is much more precise. In this figure, we label the X axis as a log scale over time. This is because analysis P took over two minutes to complete, in contrast to the longest-running refinement method, which took less than 6 seconds. The relationship between the refinement analyses is similar to the previous configuration. The key observation here is that, again, all three refinement analyses achieve within $\sim 3 \cdot e^{-5}$ of P, but this time in 4% of the time (as opposed to $\frac{1}{4}$ in the previous configuration).

Figure 5.6(d) makes more explicit the relationship between refinements (CE, S, CE+S) and P. We fix $n = 50,000$ (the maximum) here, and $p = 64$ (the maximum). We can see that as query complexity goes up, P gets exponentially slower, while CE, S, and CE+S slow at a much lower rate, while retaining (per the previous graphs) similar precision.

5.7.3 Evacuation Problem

We conclude this section by briefly discussing an analysis of an execution of the resource allocation algorithm of Figure 5.2. In our experiment, we set the number of ships to be three, where two were in range $d = 300$ of the evacuation site, and their sum-total berths (500) were sufficient to satisfy demand at the site (also 500). For our analysis refinements we set $n = 1000$. Running the algorithm, a total of seven

Table 5.1: Analyzing a 3-ship resource allocation run

Resource Allocation (3 ships)		
Analysis	Time (s)	Vulnerability
P	Timeout (5 min)	N/A
I	0.516	1
CE	16.650	$1.997 \cdot 10^{-24}$
S	1.487	$1.962 \cdot 10^{-24}$
CE+S	17.452	$1.037 \cdot 10^{-24}$

pairs of *Nearby* and *Capacity* queries were issued. In the end, the algorithm selects two ships to handle the evacuation.

Table 5.1 shows the time to execute the algorithm using the different analysis methods, along with the computed vulnerability—this latter number represents the coordinator's view of the most likely nine-tuple of the private data of the three ships involved (x coordinate, y coordinate, and capacity for each). We can see that, as expected, our refinement analyses are far more efficient than baseline P, and far more precise than baseline I. The CE methods are precise but slower than S. This is because of the need to count the number of points in the DNF of the concolic path conditions, which is expensive.

Discussion The queries considered in Figure 5.6 have two features that contribute to the effectiveness of our refinement methods. First, they are defined over large domains, but return `true` for only a small subset of those values. For larger subsets of values, the benefits of sampling may degrade, though concolic execution should still provide an improvement. Further experiments are needed to explore such scenarios. Second, the example in Figure 5.6 contains short but complex queries. A result of this query structure is that abstract interpretation with polyhedra is expensive but sampling can be performed efficiently. The evacuation problem results in Table 5.1 provide some evidence that the benefits of our techniques also apply to longer queries. However it may still be possible to construct queries where the gap in runtime between

polyhedral analysis and sampling is smaller, in which case sampling would provide less improvement.

5.8 Related Work

Quantifying Information Flow. There is a rich research literature on techniques that aim to *quantify* information that a program may release, or has released, and then use that quantification as a basis for policy. One question is what measure of information release should be used. Past work largely considers information theoretic measures, including *Bayes vulnerability* [163] and *Bayes risk* [41], *Shannon entropy* [159], and *guessing entropy* [116]. The *g-vulnerability* framework [4] was recently introduced to express measures having richer operational interpretations, and subsumes other measures.

Our work focuses on Bayes Vulnerability, which is related to min-entropy. Vulnerability is appealing operationally: As Smith [163] explains, it estimates the risk of the secret being guessed in one try. While challenging to compute, this approach provides meaningful results for non-uniform priors. Work that has focused on other, easier-to-compute metrics, such as Shannon entropy and channel capacity, require deterministic programs and priors that conform to uniform distributions [117, 8, 127, 100, 101]. The work of Klebanov [94] supports computation of both Shannon entropy and min-entropy over deterministic programs with non-uniform priors. The work takes a symbolic execution and program specification approach to QIF. Our use of concolic execution for counting polyhedral constraints is similar to that of Klebanov. However, our language supports probabilistic choice and in addition to concolic execution we also provide a sampling technique and a sound composition. Like Mardziel et al. [115], we are able to compute the worst-case vulnerability, i.e., due to a particular output, rather than a *static* estimate, i.e., as an expectation over all possible

outputs. Köpf and Basin [99] originally proposed this idea, and Mardziel et al. were the first to implement it, followed by several others [28, 78, 103].

Köpf and Rybalchenko [100] (KR) also use sampling and concolic execution to statically quantify information leakage. But their approach is quite different from ours. KR uses sampling of a query's inputs in lieu of considering (as we do) all possible outputs, and uses concolic execution with each sample to ultimately compute Shannon entropy, by underapproximation, within a confidence interval. This approach benefits from not having to enumerate outputs, but also requires expensive model counting *for each sample*. By contrast, we use sampling and concolic execution *from the posterior* computed by abstract interpretation, using the results to boost the lower bound on the size/probability mass of the abstraction. Our use of sampling is especially efficient, and the use of concolic execution is completely sound (i.e., it retains 100% confidence in the result). As with the above work, KR requires deterministic programs and uniform priors.

Probabilistic Programming Langauges. A probabilistic program is essentially a lifting of a normal program operating on single values to a program operating on distributions of values. As a result, the program represents a joint distribution over its variables [77]. As discussed in this paper, quantifying the information released by a query can be done by writing the query in a probabilistic programming language (PPL) and representing the uncertain secret inputs as distributions. Quantifying release generally corresponds to either the maximum likelihood estimation (MLE) problem or the maximum a-posteriori probability (MAP) problem. Not all PPLs support computation of MLE and MAP, but several do.

PPLs based on partial sampling [75, 133] or full enumeration [139] of the state space are unsuitable in our setting: they are either too inefficient or too imprecise. PPLs based on algebraic decision diagrams [43], graphical models [118], and factor graphs [33, 136, 120] translate programs into convenient structures and take advantage

of efficient algorithms for their manipulation or inference, in some cases supporting MAP or MLE queries (e.g. [135, 129]). PSI [66] supports exact inference via computation of precise symbolic representations of posterior distributions, and has been used for dynamic policy enforcement [103]. Guarnieri et al. [78] use probabilistic logic programming as the basis for inference; it scales well but only for a class of queries with certain structural limits, and which do not involve numeric relationships.

Our implementation for probabilistic computation and inference differs from the above work in two main ways. Firstly, we are capable of *sound* approximation and hence can trade off precision for performance, while maintaining soundness in terms of a strong security policy. Even when using sampling, we are able to provide precise confidence measures. The second difference is our *compositional* representation of probability distributions, which is based on numerical abstractions: intervals [46], octagons [119], and polyhedra [48]. The posterior can be easily used as the prior for the next query, whereas prior work would have to repeatedly analyze the composition of past queries.

A few other works have also focused on abstract interpretation, or related techniques, for reasoning about probabilistic programs. Monniaux [123] defines an abstract domain for distributions. Smith [166] describes probabilistic abstract interpretation for verification of quantitative program properties. Cousot [49] unifies these and other probabilistic program analysis tools. However, these do not deal with sound distribution conditioning, which is crucial for belief-based information flow analysis. Work by Sankaranarayanan et al [153] uses a combination of techniques from program analysis to reason about distributions (including abstract interpretation), but the representation does not support efficient retrieval of the maximal probability, needed to compute vulnerability.

Chapter 6

Conclusion

Our goal was to show that secure programming can be made safer through language-based techniques for expressive, coordinated MPC; probabilistically oblivious execution; and quantitative analysis of information flow. We provided evidence in the form of Symphony, an expressive MPC language with support for coordinating many parties; $\lambda_{\textbf{Obliv}}$, a core language for oblivious computation that ensures well-typed programs are probabilistically oblivious; and an application of dynamic analysis techniques to an existing system for bounding information leakage, providing a better balance of precision and performance. But, of course, the quest for even better languages for secure computation continues. We conclude with some promising directions for future work below.

Synthesis of Wysteria, $\lambda_{\textbf{MPC}}$, and $\lambda_{\textbf{Obliv}}$ Ultimately, we would like an MPC language with the flexible coordination of $\lambda_{\textbf{MPC}}$ and static checking of deadlock-freedom and probabilistic obliviousness. Roughly speaking, this would require generalizing Wysteria's [143] type system to support the additional features provided by $\lambda_{\textbf{MPC}}$ and then unifying the resulting language with $\lambda_{\textbf{Obliv}}$. Unifying the syntax and operational semantics should present no major issues, since $\lambda_{\textbf{Obliv}}$ was designed to be MTO and can therefore be expressed in the circuit model. However, unifying the type

systems is likely to be much more challenging. The proof of PMTO for $\lambda_{\mathbf{Obliv}}$ relies on random values being the only kind of value which can be coerced from secret to public. An MPC language provides a general-purpose declassification mechanism (i.e. decryption) and so the definition and proof of PMTO must be adapted to account for declassifications.

Resource Awareness for MPC Languages MPC programs are expensive and their performance can be impacted dramatically by both the features of the underlying protocol and innocuous variations in the program itself. For example, a 2-party MPC program may wish to use Yao's protocol if it is being deployed on a high-latency network but would probably choose GMW for deployments on a network with very low latency. Additionally, a non-expert programmer might use multiplexors in places where logical operators would suffice, leading to bloated circuits. There may be opportunities to integrate *resource-based* reasoning into the language to help programmers optimize performance. For example, a language like Resource-Aware ML [85] could provide a good starting place.

High-Performance Compilation $\lambda_{\mathbf{MPC}}$ is implemented as an interpreter, and incurs non-negligible overhead as a result. In addition to being a significant engineering challenge, there are also open research problems that must be solved to achieve peak performance while also providing flexible features. For example, most MPC frameworks based on secret sharing [92, 5] execute circuits layer-by-layer ensuring that gates on a particular layer are executed in parallel. However, as observed by Braun et al. [36], this approach is not always optimal in practice despite being asymptotically optimal. Optimally executing a circuit depends both on the network conditions (e.g. bandwidth, latency) as well as the structure of the circuit. This challenge is exacerbated by reactive MPC, which complicates any potential analysis of optimal circuit execution by demanding that circuits be executed as they are being constructed. We

leave the design and implementation of a high-performance compiler for a flexible, coordinated MPC language like $\lambda_{\mathbf{MPC}}$ to future work.

User Studies of MPC Languages We claim in this proposal that the *choreographic*, or SIMD, approach to MPC programming renders them more usable and less error-prone. We justify that claim by showing that such a design can eliminate deadlock that can occur in other MPC languages. However, choreographic languages [50, 51, 137, 124] are not the only way of achieving deadlock-free distributed programming. Another popular alternative are process languages [152, 83] which can be guaranteed deadlock-free using session types [88]. It would be interesting to evaluate the usability of various approaches to distributed programming in the context of MPC.

Appendix A

SYMPHONY: Architecture and Proofs

A.1 FFI and Resource Management

The Haskell interpreter for SYMPHONY implements MPC using the SYMPHONY runtime library. The SYMPHONY runtime library implements MPC using the EMP and MOTION libraries. Each of these layers (SYMPHONY interpreter, SYMPHONY runtime, and EMP/MOTION) interact via FFI, using opaque foreign pointers to represent values in the next layer.

In Haskell, the raw foreign pointers are wrapped with the ForeignPtr type, which executes an associated *finalizer* (i.e. a Haskell function) on the raw foreign pointer when it is garbage collected. We use this finalizer to call (via FFI) an appropriate destructor function which is provided by SYMPHONY runtime.

In Rust, the raw foreign pointers are wrapped with `newtype`-style Rust structures. These Rust structures contain an explicit implementation of the Drop trait, calling the `drop` function on the structure when it is dropped. This is analogous to the `ForeignPtr` in Haskell, except that Rust can insert calls to `drop` statically rather than relying on garbage collection. Just as in Haskell, the `drop` function calls (via FFI) an appropriate destructor function, which is provided by EMP and MOTION.

Finally, we implement FFI interfaces for both EMP and MOTION which expose constructor and destructor functions which perform heap allocation and deallocation of the requisite C++ objects.

Putting all this together, when an encrypted value is garbage collected by Haskell the `ForeignPtr` will call the appropriate destructor defined by SYMPHONY runtime. Then, that destructor function will drop the value which will call the appropriate destructor defined by EMP or MOTION. Finally, the destructor of EMP or MOTION exposed by the FFI will free the C++ object using the `delete` keyword.

These approaches to integrating software written in different languages are largely standard. We chose to implement the enhancements in SYMPHONY runtime as a separate library to optimize performance. We chose Rust specifically because it has excellent libraries, tooling, and documentation. The SYMPHONY runtime is written in idiomatic Rust, meaning that it may serve as an artifact of independent interest for researchers who need access to MPC protoocols with support for delegation, resharing, and reactive MPC.

A.2 Metatheory

A.2.1 Proof Sketches for Correspondence Theorems

To prove theorems Theorem 3.4.1, Theorem 3.4.2 and Theorem 3.4.3 given in Section 3.4, we first formalize key definitions.

Definition A.2.1 (Divergence)**.** *A single-threaded configuration ς is divergent if for all ς' where $\varsigma \longrightarrow^* \varsigma'$, there exists ς'' s.t. $\varsigma' \longrightarrow \varsigma''$. (And likewise for distributed configurations C and transitions $\rightsquigarrow^*$.)*

Definition A.2.2 (Locally stuck)**.**

$$
\begin{aligned}
C \text{ is locally stuck} &\triangleq \exists A \text{ s.t. } C(A) = \dot{\varsigma} \\
\textit{and where}\quad & \dot{\varsigma} \text{ is not a terminal state} \\
& \dot{\varsigma}.e \notin \{\texttt{share}[\_ \to \_]\ \_, \texttt{reveal}[\_ \to \_]\ \_\} \\
& \dot{\varsigma} \not\longrightarrow_A \\
\textit{or}\quad & \dot{\varsigma}.e \in \{\texttt{share}[x_1 \to x_2]\ x_3, \texttt{reveal}[x_1 \to x_2]\ x_3\} \\
& p = \dot{\varsigma}.\dot{\gamma}(x_1) \qquad m = \dot{\varsigma}.m \\
& q = \dot{\varsigma}.\dot{\gamma}(x_2) \qquad m \neq p \cup q
\end{aligned}
$$

Now we establish a number of key lemmas. Our proof approach for Theorem 3.4.1 largely follows the proof approach from Wysteria [143], and our proof approach for Theorem 3.4.2 and Theorem 3.4.3—while novel—are straightforward proofs by case analysis and inductive reasoning on the recursive syntax of configurations and inductively defined relations $\longrightarrow$, $\rightsquigarrow$ and $\longrightarrow_A$. In this section we show the high level proof approach.

First, we establish determinism for the single-threaded semantics and confluence for the distributed semantics:

Lemma A.2.0.1 (ST Determinism)**.** *If* $\varsigma \longrightarrow \varsigma_1$ *and* $\varsigma \longrightarrow \varsigma_2$ *then* $\varsigma_1 = \varsigma_2$.

Proof. Case analysis on derivations $\varsigma \longrightarrow \varsigma_1$ and $\varsigma \longrightarrow \varsigma_2$. □

Lemma A.2.0.2 (D Confluence)**.** *If* $C \rightsquigarrow^* C_1$ *and* $C \rightsquigarrow^* C_2$ *then* $C_1 \rightsquigarrow^* C_3$ *and* $C_2 \rightsquigarrow^* C_3$ *for some* C_3.

Proof. We first prove a diamond property sublemma that shows if $C \rightsquigarrow C_1$, $C \rightsquigarrow C_2$ and $C_1 \neq C_2$, then $C_1 \rightsquigarrow C_3$ and $C_2 \rightsquigarrow C_3$ for some C_3, which is proved by case analysis on derivations $C \rightsquigarrow C_1$ and $C \rightsquigarrow C_2$. Confluence is established as a classic results whereby transition systems which satisfy the diamond property are also confluent, the proof of which is by induction on derivations $C \rightsquigarrow^* C_1$ and $C \rightsquigarrow^* C_2$ and appealing to the diamond property in the base cases. □

Next, we establish forward simulation between terminal states and semantics:

Lemma A.2.0.3 (ST Forward Simulation)**.**

1. *If* ς *is terminal then* $\varsigma\lightning$ *is terminal*
2. *If* ς *is stuck then* $\varsigma\lightning$ *is locally stuck*
3. *If* $\varsigma \longrightarrow^* \varsigma'$ *then* $\varsigma\lightning \rightsquigarrow^* \varsigma'\lightning$.

Proof.

1. Case analysis on ς
2. Case analysis on ς
3. Induction on steps in $\varsigma \longrightarrow^* \varsigma'$ and case analysis on intermediate derivations $\varsigma \longrightarrow \varsigma''$.

□

Theorem 3.4.1 then follows from these lemmas:

Proof of ST/D Terminal Correspondence. The forward direction is equivalent to showing $\varsigma \longrightarrow^* \varsigma'$ and ς' terminal implies $\varsigma\lightning \longrightarrow^* \varsigma'\lightning$ and $\varsigma'\lightning$ terminal, which follows from Lemma A.2.0.3.

The backward direction is equivalent to showing $\varsigma\lightning \rightsquigarrow^* C$ and C terminal implies $\varsigma \longrightarrow^* \varsigma'$ for some ς' where ς' terminal and $C = \varsigma'\lightning$. By Lemma A.2.0.3 and Lemma A.2.0.2 we know that if ς diverges then $\varsigma\lightning$ must diverge, and therefore under the assumption that $\varsigma\lightning$ converges, we know must converge, so $\varsigma \longrightarrow^* \varsigma'$ for some terminal state ς'. By Lemma A.2.0.3 we know $\varsigma\lightning \rightsquigarrow^* \varsigma'\lightning$, and by Lemma A.2.0.2 we know $C = \varsigma'\lightning$. □

Our proof of Theorem 3.4.2 also follows from the lemmas and theorem proven thus far:

Proof of ST/D Strong Asymmetric Non-terminal Correspondence.

1. By Theorem 3.4.1 we know ς doesn't reach a terminal state, so it either diverges or converges to a stuck state. Consider each case. Assume ς diverges, then we know by Lemma A.2.0.3 we know that there exists a distributed trace that also diverges. By Lemma A.2.0.2 applied to the stuck distributed state, the divergent distributed state (just established), and $\varsigma\lightning$ as the common ancestor, we know the stuck distributed state can make progress towards a divergent one, which is a contradiction—so this subcase can never happen. The other subcase is when ς reaches a stuck state, which trivially satisfies the goal.

2. Because $\rightsquigarrow$ is confluent by Lemma A.2.0.2, $\varsigma\lightning$ must either converge to a terminal state, converge to a stuck state, or diverge. (E.g., it impossible for $\varsigma\lightning \rightsquigarrow^* \varsigma'$ where ς' is stuck, and for $\varsigma\lightning \rightsquigarrow^* \varsigma''$ where ς'' can continue to transition without ever reaching a stuck or terminal state.) If $\varsigma\lightning$ converged then by Theorem 3.4.1, which would reach a contradiction. If $\varsigma\lightning$ reached a stuck state, then so would ς by (1) of this theorem, which would reach a contradiction. Therefore, $\varsigma\lightning$ must diverge.

□

We prove one final lemma before proving our third theorem:

Lemma A.2.0.4 (D Local Stuck Preservation)**.** *If C is locally stuck and $C \rightsquigarrow^* C'$ then C' is locally stuck.*

Proof. Induction on the number of steps in $\rightsquigarrow^*$, and case analysis on intermediate derivations $C \rightsquigarrow C''$. □

Our proof of Theorem 3.4.3 then uses the prior lemma:

Proof of ST/D Soundness for Stuck States. We assume $\varsigma \longrightarrow^* \varsigma'$ where ς' is stuck and some C where $\varsigma\lightning \rightsquigarrow C$. We must show there exists C' s.t. $C \rightsquigarrow C'$ and C' locally stuck. By Lemma A.2.0.3 we know $\varsigma\lightning \rightsquigarrow^* \varsigma'\lightning$ and $\varsigma'\lightning$ is locally stuck. By confluence we have there exists C' s.t. $C \rightsquigarrow^* C'$ and $\varsigma'\lightning \rightsquigarrow^* C'$. By Lemma A.2.0.4 with $\varsigma\lightning$ as the common ancestor we have C' locally stuck. □

Theorem 3.4.1 captures the same metatheoretical properties proved of prior work (Wysteria [143]), whereas Theorems 3.4.2 and 3.4.3 are refinements of divergence-soundness and stuck-state-soundness results novel to our work.

A.2.2 Detailed Proofs for Key Lemmas

In this section, we prove the key meta-theoretic properties of the distributed semantics, namely forward simulation (Appendix A.2.2) and confluence (Appendix A.2.2), along with their corollaries. The full DS-semantics rules are given in **??**.

Forward Simulation

The key lemma for proving simulation states that if global single-threaded configuration ς steps to ς', then the slicing of ς steps to ς' over multiple steps of the multi-threaded semantics. The basic structure of the proof is, based on the form of step from global configuration ς, to construct a sequence of distributed steps that each updates the local configuration of some party in the mode of ς. For non-atomic expressions, there is exactly one step for every party in the mode; the most interesting case are global steps that are applications SS-Par: these are simulated by a sequence of steps which may be built from applications of SS-Par themselves *or* SS-Empty. For expressions that evaluate an atom and bind the result, there is a single step, performed by all parties.

Lemma A.2.0.5 (Forward Simulation-Step)**.** *If* $\varsigma \to \varsigma'$*, then* $\varsigma\lightning \rightsquigarrow^* \varsigma'\lightning$.

Proof. $\varsigma \to \varsigma'$, by assumption. Let $(m, \gamma, \delta, \kappa, e) = \varsigma$ and let $(m', \gamma', \delta', \kappa', e') = \varsigma'$. Proceed by cases on the form of the evidence of $\varsigma \to \varsigma'$:

ST-Case-Inj $\varsigma\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow \varsigma'\lightning$, where each C_i is $\varsigma\lightning|_{[0,i]} \uplus \varsigma'\lightning|_{[i+1,|m|]}$ (where $C|_I$ denotes distributed configuration C restricted to parties at indices I).

The proof that each C_i steps to C_{i+1} is as follows. Apply DS-Step, with $\dot{\varsigma}$ as the configuration

$$m_i, \gamma, \delta, \kappa, \text{case } x\{x_1.e_1\}\{x_2.e_2\}$$

$\dot{\varsigma}'$ as the configuration

$$m_i, \{x \mapsto v\} \uplus \gamma, \delta, \kappa, e_j$$

where $\gamma(x) \swarrow_{m_i} = (\iota_j v)@\{m_i\}$ and $C_i|_{[0,i-1],[i+1,|m|]}$ as C. $\dot{\varsigma} \longrightarrow_i \dot{\varsigma}'$ by DS-Case-Inj. C_{i+1} is $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \dot{\varsigma}'\} \uplus C_i|_{[i+1,|m|]}$.

The proofs for evidence constructed from rules DS-Case-PSet-Emp and DS-Case-PSet-Cons are similar. The only distinction is that the updated local configuration in each distributed configuration C_{i+1} is formed by updating the subject of expression to e_1 in the case of Rule DS-Case-PSet-Emp and e_2 in the case of Rule DS-Case-PSet-Cons. Additionally, in the case of Rule DS-Case-PSet-Cons, the local state is updated to bind variables x_2 and x_3 to the deconstructed principal and remaining set of principals.

ST-Par $\varsigma\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow \varsigma'\lightning$, where each C_i is $\varsigma\lightning|_{[0,i]} \uplus \varsigma'\lightning|_{[i+1,|m|]}$.

The proof that each C_i steps to C_{i+1} is as follows. If $m_i \in p$, then apply DS-Step, with $\dot{\varsigma}$ as the configuration

$$m_i, \gamma, \delta, \kappa, \texttt{par}\ p\ e$$

$\dot{\varsigma}'$ as the configuration

$$m_i, \gamma, \delta, \kappa, e$$

and $C_i|_{[0,i-1],[i+1,|m|]}$ as C. $\dot{\varsigma} \longrightarrow_i \dot{\varsigma}'$ by DS-Par, because $m_i \in p$ and thus $\{m_i\} \cap p = \{m_i\} \neq \emptyset$.

If $m_i \notin p$, then let $\dot{\varsigma}' = \dot{\varsigma}$.

In both cases, C_{i+1} is $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \dot{\varsigma}'\} \uplus C_i|_{[i+1,|m|]}$.

ST-ParEmpty $\varsigma\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow \varsigma'\lightning$, where each C_i is $\varsigma\lightning|_{[0,i]} \uplus \varsigma'\lightning|_{[i+1,|m|]}$.

The proof that each C_i steps to C_{i+1} is as follows. Apply DS-Step, with $\dot{\varsigma}$ as the configuration

$$m_i, \gamma, \delta, \kappa, \texttt{par}\ p\ e$$

$\dot{\varsigma}'$ as the configuration

$$m_i, \{x \mapsto \bigstar\} \uplus \gamma, \delta, \kappa, x$$

and $C_i|_{[0,i-1],[i+1,|m|]}$ as C. $\dot{\varsigma} \longrightarrow_i \dot{\varsigma}'$ by DS-ParEmpty, because $m \cap p = \emptyset$ by the fact that $c \to c$ is an application of ST-ParEmpty; thus $\{m_i\} \cap p = \emptyset$. C_{i+1} is $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \dot{\varsigma}'\} \uplus C_i|_{[i+1,|m|]}$.

ST-App $\varsigma\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow \varsigma'\lightning$, where each C_i is $\varsigma\lightning|_{[0,i]} \uplus \varsigma'\lightning|_{[i+1,|m|]}$.

The proof that each C_i steps to C_{i+1} is as follows. Let $\langle \lambda_z x.e', \gamma' \rangle @ m = v_1 = \gamma(x_1)$, which holds in the case that $c \to c'$ is an application of ST-App.

Apply DS-Step, with $\dot{\varsigma}$ as the configuration

$$m_i, \gamma, \delta, \kappa, x_1\ x_2$$

$\dot{\varsigma}'$ as the configuration

$$m_i, \{z \mapsto v_1, x_2 \mapsto \gamma(x_2)\} \uplus \gamma, \delta, \langle \texttt{let}\ x\ =\ \_\ \texttt{in}\ e_2 \mid \gamma \rangle :: \kappa, e'$$

and $C_i|_{[0,i-1],[i+1,|m|]}$ as C. $\dot{\varsigma} \longrightarrow_i \dot{\varsigma}'$ by DS-App. C_{i+1} is $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \dot{\varsigma}'\} \uplus C_i|_{[i+1,|m|]}$.

ST-LetPush $\varsigma\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow \varsigma'\lightning$, where each C_i is $\varsigma\lightning|_{[0,i]} \uplus \varsigma'\lightning|_{[i+1,|m|]}$.

The proof that each C_i steps to C_{i+1} is as follows. Apply DS-Step, with $\dot{\varsigma}$ as the configuration

$$m_i, \gamma, \delta, \kappa, \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2$$

$\dot{\varsigma}'$ as the configuration

$$m_i, \gamma, \delta, \langle \texttt{let}\ x\ =\ \_\ \texttt{in}\ e_2 \mid \rangle :: \kappa, e_1$$

and $C_i|_{[0,i-1],[i+1,|m|]}$ as C. $\dot{\varsigma} \longrightarrow_i \dot{\varsigma}'$ by DS-LetPush. C_{i+1} is $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \dot{\varsigma}'\} \uplus C_i|_{[i+1,|m|]}$.

ST-LetPop e is some atom a, δ and a step to δ' and some value v under γ in mode m, and $\kappa = \langle \texttt{let}\ x\ =\ \_\ \texttt{in}\ e' \mid m', \gamma'' \rangle :: \kappa'$, and $\gamma' = \{x \mapsto v\} \uplus \gamma''$, by assumption.

Proceed by cases on the fact that δ and a step to δ' and some value v under γ in mode m. **Subcase: solo atom** In the case that the evaluation is an application of ST-Int, ST-Var, ST-Fun, ST-Inj, ST-Pair, ST-Proj, ST-Ref, ST-Deref, ST-Assign, ST-Fold, ST-Unfold, ST-Read, ST-Write, ST-Embed, and ST-Star, $\varsigma\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow \varsigma'\lightning$, where each C_i is $\varsigma\lightning|_{[0,i]} \uplus \varsigma'\lightning|_{[i+1,|m|]}$. The proof that each C_i steps to C_{i+1} is as follows. Apply DS-Step, with $\dot{\varsigma}$ as the configuration

$$m_i, \gamma, \delta, \langle \texttt{let}\ x\ =\ \_\ \texttt{in}\ e' \mid m', \gamma'' \rangle :: \kappa', a$$

$\dot{\varsigma}'$ as the configuration

$$m_i, \{x \mapsto v\} \uplus \gamma, \delta', \kappa', e'$$

and $C_i|_{[0,i-1],[i+1,|m|]}$ as C. $\dot{\varsigma} \longrightarrow_i \dot{\varsigma}'$ by DS-LetPop. C_{i+1} is $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \dot{\varsigma}'\} \uplus C_i|_{[i+1,|m|]}$.

Subcases: binary operation over clear data The subcases in which evaluation is an application of ST-Binop, a is of the form $x_1 \oplus x_2$, $\dot{i_1}@m = \gamma(x_1) \swarrow_m$, and $\dot{i_2}@m = \gamma(x_1) \swarrow_m$ or in which evaluation is an application of ST-PSet-Binop (i.e., the computation is an binary operation over clear data) is directly similar to the previous subcase.

Subcase: mux on clear data The subcase in which evaluation is an application of ST-Mux, a is of the form $\texttt{mux if } x_1 \texttt{ then } x_2 \texttt{ else } x_3$, $\dot{i_1}@m = \gamma(x_1) \swarrow_m$, $\dot{i_2}@m = \gamma(x_2) \swarrow_m$, and $\dot{i_3}@m = \gamma(x_3) \swarrow_m$ is directly similar to the previous subcases.

Subcase: binary operation on encrypted data For the subcase in which evaluation is an application of ST-Binop, a is of the form $x_1 \oplus x_2$, $\dot{i}_1^{\mathsf{enc}\#m}@m = \gamma(x_1) \swarrow_m$, and $\dot{i}_2^{\mathsf{enc}\#m}@m = \gamma(x_1) \swarrow_m$ (i.e., the computation is an binary operation over encrypted data), $\varsigma \lightning$ steps to $\varsigma' \lightning$ by application of DS-Step, with

$$m, \gamma, \delta, \kappa, x_1 \oplus x_2$$

as $\dot{\varsigma}$,

$$m, \{x \mapsto v\} \uplus \gamma, \delta, \kappa, x$$

as $\dot{\varsigma}'$, and $\varsigma \lightning|_{\text{parties} \setminus m}$ as C. $\dot{\varsigma}$ steps to $\dot{\varsigma}'$ by ST-Binop.

Subcase: mux on encrypted data The subcase in which evaluation is an application of ST-Mux, a is of the form $\texttt{mux if } x_1 \texttt{ then } x_2 \texttt{ else } x_3$, $\dot{i}_1^{\mathsf{enc}\#m}@m = \gamma(x_1) \swarrow_m$, $\dot{i}_2^{\mathsf{enc}\#m}@m = \gamma(x_2) \swarrow_m$, and $\dot{i}_3^{\mathsf{enc}\#m}@m = \gamma(x_3) \swarrow_m$ is directly similar

to the previous subcase.

Subcases: synchronization The subcases in which evaluation is an application of ST-Share or ST-Reveal are directly similar to the previous two subcaes, in that they are simulated by a single step of the distributed semantics.

□

The proof of weak forward simulation follows directly from Lemma A.2.0.5.

Lemma A.2.0.6 (ST Weak Forward Simulation)**.** *If* $\varsigma \longrightarrow^* \varsigma'$ *and* ς' *is terminal, then* $\varsigma\lightning \rightsquigarrow^* \varsigma'\lightning$ *and* $\varsigma'\lightning \not\rightsquigarrow$*.*

Proof. The claim holds by induction on the multistep judgment $\varsigma \longrightarrow^* \varsigma$.

Empty If the trace is empty, then $\dot{\varsigma}$ is $\dot{\varsigma}'$. $\varsigma\lightning$ multi-steps to $\varsigma\lightning$ over the empty sequence of steps.

Non-empty If the trace is of the form $\varsigma \to \varsigma'' \to^* \varsigma'$, then $\varsigma\lightning \rightsquigarrow^* \varsigma''$ by Lemma A.2.0.5 and $\varsigma''\lightning \rightsquigarrow^* \varsigma'\lightning$ by the inductive hypothesis. $\varsigma\lightning \rightsquigarrow^* \varsigma\lightning$ by the fact that the concatenation of two traces is a trace.

□

Confluence and End-State Determinism

In order to prove the Diamond Property, we will first claim and prove a lemma that establishes that distinct sub-configurations that can step within each step of a distributed configuration in fact update the local configurations of disjoint sets of parties.

Lemma A.2.0.7. *For all distributed configurations* C*,* C_0*, and* C_1 *and all non-halting distributed configurations* C_0' *and* C_1' *such that*

$$C = C_0' \uplus C_0 = C_1' \uplus C_1$$

one of the following cases holds:

1. $C'_0 = C'_1$ *and* $C_0 = C_1$*;*

2. *the domains of* C'_0 *and* C'_1 *are disjoint.*

Proof. Proceed by cases on whether the domains of C'_0 and C'_1 are disjoint. If so, then the second clause of the claim is satisfied.

Otherwise, there is some party m_i in the domains of both C'_0 and C'_1. The domains of C'_0 and C'_1 are the same, by cases on the active expression e_i in the local configuration located at m_i: if e_i is a non-atom, a variable occurrence, an integer literal, a binary operation over integers, a binary operation over sets of principals, a multiplex, a pair creation, a pair projection, a sum injection, a function creation, a reference creation, a dereference, a reference assignment, a recursive type introduction, a read, or a write then the domains are singletons. Thus the domains are the same, because they are singletons that overlap.

In the case that the expression shares a value from p to q, the steps from C'_0 and C'_1 are applications of DS-Share, which has a premise that the mode is $p \cup q$; thus, the domains of C'_0 and C'_1 are the identical set of parties $p \cup q$.

In the case that the expression reveals the value bound to variable x to parties q, the steps from C'_0 and C'_1 are applications of DS-Reveal, which has a premise that the value bound to x is encrypted for parties p, and that the active parties are $p \cup q$; thus, the domains of C'_0 and C'_1 are the same set of parties $p \cup q$. C_0 and C_1 are thus the same, given they are the restrictions of C to the complements of the domains of C'_0 and C'_1, respectively. □

Using Lemma A.2.0.7, we can prove the Diamond Property for the transition relation over multi-threaded configurations.

Proof. There are distributed configurations $C_{0,0}$, $C_{0,1}$, $C_{1,0}$, and $C_{1,1}$ such that

$$C_{0,0} \uplus C_{0,1} = C = C_{1,0} \uplus C_{1,1}$$

and

$$\begin{aligned} C_0 &= C_{0,0} \uplus C'_{0,1} \\ C_1 &= C_{1,0} \uplus C'_{1,1} \end{aligned}$$

with $C_{0,1} \rightsquigarrow C'_{0,1}$ and $C_{1,1} \rightsquigarrow C'_{1,1}$, by inverting the facts that C steps to C_0 and C steps to C_1. Proceed by cases on the application of Lemma A.2.0.7 to C, $C_{0,0}$, $C_{0,1}$, $C_{1,0}$, and $C_{1,1}$:

Identical It follows immediately that $C_{0,0} = C_{1,0}$. Furthermore, it follows from a direct analysis of the multi-threaded transition relation that $C'_{0,1} = C'_{1,1}$. Thus

$$C_0 = C_{0,0} \uplus C'_{0,1} = C_{1,0} \uplus C'_{1,1} = C_1$$

by congruence. Thus for $C' = C'_0 = C'_1$, both $C \rightsquigarrow C'_0$ and $C \rightsquigarrow C'_1$.

Disjoint Let C'' be C restricted to parties in $C_{0,0}$ and $C_{1,0}$ and let

$$C' = C'' \uplus C'_{0,1} \uplus C'_{1,1}$$

C' is well-defined because the domains of $C'_{0,1}$ and $C'_{1,0}$, are the domains of $C_{0,1}$ and $C_{1,1}$, which are disjoint by assumption of this clause.

$C_0 \rightsquigarrow C'$ by cases on the fact that $C_0 \rightsquigarrow C'_0$: in each case, adjust the evidence to use $C'' \uplus C_1$ as the distributed configuration that remains unchanged and is joined with C_0. $C_1 \rightsquigarrow C'$ by a symmetric argument.

□

Given that the distributed semantics satisfies the diamond property, confluence (Lemma A.2.0.8) is a direct consequence of fundamental properties of general transition and rewrite systems.

Lemma A.2.0.8 (DS Multi-step Confluence)**.** *If* $C \rightsquigarrow^* C_1$ *and* $C \rightsquigarrow^* C_2$ *then there exists* C_3 *s.t.* $C_1 \rightsquigarrow^* C_3$ *and* $C_2 \rightsquigarrow^* C_3$*.*

Proof. Apply the fact that any binary relation that satisfies the Diamond property satisfies confluence [7] to the Diamond Property for the distributed step relation. □

An direct corollary of confluence is that all halting states reached from the same state are the same.

Corollary A.2.0.1 (DS End-state Determinism)**.** *If* $C \rightsquigarrow^* C_1$ *and* $C \rightsquigarrow^* C_2$*,* $C_1 \not\rightsquigarrow$ *and* $C_2 \not\rightsquigarrow$ *then* $C_1 = C_2$*.*

Proof. There is some distributed configuration C' such that $C_1 \rightsquigarrow^* C'$ and $C_2 \rightsquigarrow^* C'$, by applying Lemma A.2.0.8 to the fact that $C \rightsquigarrow^* C_1$ and $C \rightsquigarrow^* C_2$. C' is C_1 by the fact that C_1 is halting and thus C_1 multi-steps to C' over the empty sequence of steps; C' is C_2 by a symmetric argument. Thus, C_1 is C_2. □

Appendix B

λ_{Obliv}: Definitions and Proofs

B.1 Complete PMTO Proof

In this section we give a complete proof of PMTO. First, in Section B.1.1 we present the final proof of PMTO in top-down breadth-first organization for major lemmas, and depth-first organization for sublemmas required to prove major lemmas. In many proofs we abbreviate "suffices to show" as "STS". Next, in Section B.1.3 we show complete definitions for all semantics, type rules, auxiliary metafunctions, and low-equivalence relations which are used in the proof.

The heart of the type system design is typing for flip values:

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

This invariant dictates that (1) the distribution is uniform, and (2) that it is jointly independent of all other flip values in the execution context Ψ^F, and all other secret bit values in the execution context at strictly lower region Ψ^B. Joint independence is crucial and strictly stronger than individual independence; to see this, note that

$A \perp\!\!\!\perp B$ and $A \perp\!\!\!\perp C$ does *not* imply $A \perp\!\!\!\perp B, C$, however the converse is true.

The heart of the proof is Type Preservation, and its main sublemma Type Preservation Redex. The key semantic property of mux operations used in those lemmas is Cond Stability.

B.1.1 Theorems and Lemmas

The main metatheory result for $\lambda_{\textbf{Obliv}}$ is PMTO. The proof follows from major sublemmas.

Theorem B.1.1 (PMTO)**.**

Probabilistic equality modulo adversary observability for source expressions is preserved by the ground truth semantics.

If: e_1 *and* e_2 *are closed source expressions*

And: $\vdash e_1 : \tau$ *and* $\vdash e_2 : \tau$

And: $\mathrm{obs}(e_1) = \mathrm{obs}(e_2)$

Then:

(1) $\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_1)$ *and* $\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_2)$ *are defined*

(2) $\widetilde{\mathrm{obs}}(\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_1)) = \widetilde{\mathrm{obs}}(\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e_2))$

Proof.

(1) is by Progress (Ground Truth)

(2) is by the following:

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

□

PMTO Proof Key Lemmas

Progress (Ground Truth)

Lemma B.1.1.1 (Progress (Ground Truth))**.**

Progress holds for the ground truth semantics.

If: $\vdash \varsigma$

Then: $\mathrm{nstep}_{\mathcal{D}}(N, \varsigma)$ *is total*

Proof. Induction on N and Progress (Ground Truth) Single □

Lemma B.1.1.2 (Progress (Ground Truth) Single)**.**

Progress holds for the ground truth semantics on a single step.

If: $\Sigma \vdash \sigma, e$

Then either:

(1) $e = v$ *for* v *a value*

(2) $e = E[e']$ *and* e' *a redex*

In both cases $\text{step}_{\mathcal{D}}(N, \sigma, e)$ *is total*

Proof. Induction on e and inversion on assumed well-typing □

Low-equivalence Completeness

Lemma B.1.1.3 (Low-equivalence Completeness (Source Expressions))**.**

Source expressions which are equal modulo adversary observation are low-equivalent.

If: e_1 *and* e_2 *are source expressions*

And: $\text{obs}(e_1) = \text{obs}(e_2)$

Then: $\lfloor e_1 \rfloor \sim \lfloor e_2 \rfloor$

Proof. Induction on e_1 and e_2, and discrimination on assumed $\text{obs}(e_1) = \text{obs}(e_2)$ □

PMTO (Mixed)

Lemma B.1.1.4 (PMTO (Mixed))**.**

Probabilistic low-equivalence for source expressions is preserved by the mixed semantics.

If: e_1 *and* e_2 *are closed source expressions*

And: $\vdash e_1 : \tau$ *and* $\vdash e_2 : \tau$

And: $e_1 \sim e_2$

Then:

(1) $\underline{\text{nstep}}(N, \varnothing, e_1)$ *and* $\underline{\text{nstep}}(N, \varnothing, e_2)$ *are defined*

(2) $\underline{\text{nstep}}(N, \varnothing, e_1) \approx_\sim \underline{\text{nstep}}(N, \varnothing, e_2)$

Proof.

(1) is by Progress (Mixed)

(2) is by induction on N

- Case $N = 0$:

  STS: $\text{return}(e_1) \approx_\sim \text{return}(e_2)$

  By Return Equivalence

- Case $N = N + 1$:

  $\underline{\text{nstep}}(N, \varnothing, e_1) \approx_\sim \underline{\text{nstep}}(N, \varnothing, e_2)$ *(IH)* (by inductive hypothesis)

  STS:

do $\underline{t}\cdot\underline{\varsigma} \leftarrow \underline{\mathrm{nstep}}(N, \varnothing, e_1)$

$\underline{\varsigma}' \leftarrow \underline{\mathrm{step}}(N+1, \underline{\varsigma})$

$\mathrm{return}(\underline{t}\cdot\underline{\varsigma}, \underline{\varsigma}')$

$\approx_\sim$

do $\underline{t}\cdot\underline{\varsigma} \leftarrow \underline{\mathrm{nstep}}(N, \varnothing, e_1)$

$\underline{\varsigma}' \leftarrow \underline{\mathrm{step}}(N+1, \underline{\varsigma})$

$\mathrm{return}(\underline{t}\cdot\underline{\varsigma}, \underline{\varsigma}')$

By Bind Equivalence, Return Equivalence and *(IH)*, STS:

- $\underline{t}_1\cdot\underline{\varsigma}_1 \sim \underline{t}_2\cdot\underline{\varsigma}_2 \implies \left[\underline{\mathrm{step}}(N+1, \underline{\varsigma}_1) \mid \Phi_1\right] \approx_\sim \left[\underline{\mathrm{step}}(N+1, \underline{\varsigma}_2) \mid \Phi_2\right]$

  where $\Phi_1 \triangleq [\underline{\mathrm{nstep}}(N, \varnothing, e_1) \doteq \underline{t}_1\cdot\underline{\varsigma}_1]$ and $\Phi_2 \triangleq [\underline{\mathrm{nstep}}(N, \varnothing, e_2) \doteq \underline{t}_2\cdot\underline{\varsigma}_2]$

By Type Preservation:

- There exists Σ_1, Σ_2, Ψ_1 and Ψ_2

  S.t. $\Phi_1, \Sigma_1 \vdash \underline{\varsigma}_1 \; ; \Psi_1$ and $\Phi_2, \Sigma_2 \vdash \underline{\varsigma}_2 \; ; \Psi_2$

Conclusion is by PMTO (Mixed) Single applied to premise and the above well-typing

□

Lemma B.1.1.5 (PMTO (Mixed) Single)**.**

Probabilistic low-equivalence for source expressions is preserved by the mixed semantics on a single step.

If: $\Phi_1, \Sigma_1 \vdash \underline{\varsigma}_1 \; ; \Psi_1$ *and* $\Phi_2, \Sigma_2 \vdash \underline{\varsigma}_2 \; ; \Psi_2$

And: $\underline{\varsigma}_1 \sim \underline{\varsigma}_2$

Then: $\left[\underline{\mathrm{step}}(N, \underline{\varsigma}_1) \mid \Phi_1\right] \approx_\sim \left[\underline{\mathrm{step}}(N, \underline{\varsigma}_2) \mid \Phi_2\right]$

Proof. By case analysis on $\underline{\varsigma}_1 \sim \underline{\varsigma}_2$ and Progress (Mixed); two cases:

(1) Case $\underline{\varsigma}_1 = \underline{\sigma}_1, \underline{v}_1$ and $\underline{\varsigma}_2 = \underline{\sigma}_2, \underline{v}_2$ for $\underline{v}_1$ and $\underline{v}_2$ values

$\underline{\text{step}}(N, \_)$ is the same as return on values

Immediate by Return Equivalence

(2) Case $\underline{\varsigma}_1 = \underline{\sigma}_1, \underline{E}_1[\underline{e}_1]$ and $\underline{\varsigma}_2 = \underline{\sigma}_2, \underline{E}_2[\underline{e}_2]$ for $\underline{e}_1$ and $\underline{e}_2$ redexes

$\underline{\sigma}_1 \sim \underline{\sigma}_2$

$\underline{e}_1 \sim \underline{e}_2$ (by Contexts Preserve Low Equivalence)

$[\underline{\text{step}}(N, \underline{\sigma}_1, \underline{e}_1) \mid \Phi_1] \approx_\sim [\underline{\text{step}}(N, \underline{\sigma}_2, \underline{e}_2) \mid \Phi_2]$ (by PMTO (Mixed) Redex)

$[\underline{\text{step}}(N, \underline{\sigma}_1, \underline{E}_1[\underline{e}_1]) \mid \Phi_1] \approx_\sim [\underline{\text{step}}(N, \underline{\sigma}_2, \underline{E}_2[\underline{e}_2]) \mid \Phi_2]$ (by Bind Equivalence, Return Equivalence and Contexts Preserve Low Equivalence)

□

Lemma B.1.1.6 (PMTO (Mixed) Redex)**.**

Probabilistic low-equivalence for source expressions is preserved by the mixed semantics on a single step for redex configurations.

If: $\underline{\varsigma}_1$ *and* $\underline{\varsigma}_2$ *are redex configurations*

And: $\Phi_1, \Sigma_1 \vdash \underline{\varsigma}_1 \,;\, \Psi_1$ *and* $\Phi_2, \Sigma_2 \vdash \underline{\varsigma}_2 \,;\, \Psi_2$

And: $\underline{\varsigma}_1 \sim \underline{\varsigma}_2$

Then: $\left[\underline{\text{step}}(N, \underline{\varsigma}_1) \;\middle|\; \Phi_1\right] \approx_\sim \left[\underline{\text{step}}(N, \underline{\varsigma}_2) \;\middle|\; \Phi_2\right]$

Proof. By inversion:

$$\frac{\underline{\sigma}_1 \sim \underline{\sigma}_2 \qquad \underline{e}_1 \sim \underline{e}_2}{\underline{\sigma}_1, \underline{e}_1 \sim \underline{\sigma}_2, \underline{e}_2}$$

Case analysis on $\underline{e}_1$ and $\underline{e}_2$ and inversion on low-equivalence judgment; all cases but two are immediate by Return Equivalence because definition of $\underline{\text{step}}$ is a return

(1) Non-immediate case $\underline{e}_1 = \texttt{cast}_P(\texttt{flipv}(\hat{b}_1))$ and $\underline{e}_2 = \texttt{cast}_P(\texttt{flipv}(\hat{b}_2))$:

By assumed well-typing:

- $\Pr\left[\hat{b}_1 \doteq \mathtt{I} \;\middle|\; \Phi_1\right] = 1/2$
- $\Pr\left[\hat{b}_2 \doteq \mathtt{I} \;\middle|\; \Phi_2\right] = 1/2$

By above facts, Bind Equivalence and because $\mathrm{return}(\mathtt{bitv}_P(\mathtt{I})) \not\sim \mathrm{return}(\mathtt{bitv}_P(\mathtt{F}))$:

$$\left[\begin{array}{l}\mathrm{do}\ b \leftarrow \hat{b}_1 \\ \quad \mathrm{return}(\mathtt{bitv}_P(b))\end{array}\;\middle|\;\Phi_1\right] \approx_\sim \left[\begin{array}{l}\mathrm{do}\ b \leftarrow \hat{b}_2 \\ \quad \mathrm{return}(\mathtt{bitv}_P(b))\end{array}\;\middle|\;\Phi_2\right]$$

(2) Non-immediate case $\underline{e}_1 = \mathtt{if}(\mathtt{bitv}_P(\hat{b}))\{\underline{e}_{11}\}\{\underline{e}_{12}\}$ and $\underline{e}_2 = \mathtt{if}(\mathtt{bitv}_P(\hat{b}))\{\underline{e}_{21}\}\{\underline{e}_{22}\}$:

By assumed well-typing:

- $\hat{b} = \mathrm{return}(b)$

By assumed low-equivalence judgment:

- $\underline{e}_{11} \sim \underline{e}_{21}$ and $\underline{e}_{12} \sim \underline{e}_{22}$

By above facts and Monad Laws:

$$\left[\begin{array}{l}\mathrm{do}\ b \leftarrow \hat{b} \\ \quad \mathrm{return}(\mathrm{cond}(b, \underline{e}_{11}, \underline{e}_{12}))\end{array}\;\middle|\;\Phi_1\right] \approx_\sim \left[\begin{array}{l}\mathrm{do}\ b \leftarrow \hat{b} \\ \quad \mathrm{return}(\mathrm{cond}(b, \underline{e}_{21}, \underline{e}_{22}))\end{array}\;\middle|\;\Phi_2\right]$$

(3) Non-immediate cases $\underline{e}_1$ and $\underline{e}_2$ let-statements or function application

By PMTO (Mixed) Substitution

□

Lemma B.1.1.7 (PMTO (Mixed) Substitution)**.**

Low-equivalence is preserved by substitution.

If: $v_1 \sim v_2$

And: $e_1 \sim e_2$

And: x *is free in* e_1 *and* e_2

Then: $[v_1/x]e_1 \sim [v_2/x]e_2$

Proof. Induction on e_1 and e_2 and inversion on assumed low equivalence □

Lemma B.1.1.8 (Contexts Preserve Low Equivalence)**.**

Low-equivalent terms have low-equivalent sub-terms, and contexts respect low-equivalence.

If: $\underline{E}_1[\underline{e}_1] \sim \underline{E}_2[\underline{e}_2]$

Then:

(1) $\underline{e}_1 \sim \underline{e}_2$

(2) $\underline{e}'_1 \sim \underline{e}'_2 \implies \underline{E}_1[\underline{e}_1] \sim \underline{E}_2[\underline{e}_2]$

Proof. Induction on $\underline{E}_1$ and $\underline{E}_2$ and inversion on assumed low equivalence

□

Lemma B.1.1.9 (Progress (Mixed))**.**

Progress holds for the mixed semantics.

If: $\Psi_c, \Phi, \Sigma \vdash \underline{\varsigma} : \tau \; ; \; \Psi$

Then: $\underline{\text{nstep}}(N, \underline{\varsigma})$ *is total*

Proof. Induction on N and Progress (Mixed) Single □

Lemma B.1.1.10 (Progress (Mixed) Single)**.**

Progress holds for the mixed semantics on a single step.

If: $\Psi_c, \Phi, \Sigma \vdash \underline{\sigma}, \underline{e} : \tau \; ; \; \Psi$

Then either:

(1) $\underline{e} = \underline{v}$ *for* $\underline{v}$ *a value*

(2) $\underline{e} = \underline{E}[\underline{e}]$ *and* $\underline{e}$ *a redex*

In both cases $\underline{\text{step}}(N, \underline{\sigma}, \underline{e})$ *is total.*

Proof. Induction on $\underline{e}$ and inversion on assumed well-typing □

Low-equivalence Soundness

Lemma B.1.1.11 (Low-equivalence Soundness)**.**

When projected, low-equivalent trace distributions have equal probability distributions modulo adversary observation.

If: $\hat{\underline{t}_1} \approx_\sim \hat{\underline{t}_2}$

Then: $\widehat{\text{obs}}(\hat{\lceil}\hat{\underline{t}_1}\hat{\rceil}) \approx_= \widehat{\text{obs}}(\hat{\lceil}\hat{\underline{t}_2}\hat{\rceil})$

Proof. Rewrite both sides by:

$$\begin{aligned} & \widehat{\text{obs}}(\hat{\lceil}\hat{\underline{t}_i}\hat{\rceil}) \\ = \; & \widehat{\text{obs}}(\text{do } \underline{t} \leftarrow \hat{\underline{t}_i} \; ; \lceil \underline{t} \rceil) \quad \wr \text{ defn. of } \hat{\lceil}\_\hat{\rceil} \int \\ = \; & \text{do } \underline{t} \leftarrow \hat{\underline{t}_i} \; ; \widehat{\text{obs}}(\lceil \underline{t} \rceil) \quad \wr \text{ defn. of } \widehat{\text{obs}} \text{ and Monad Laws } \int \end{aligned}$$

By Bind Equivalence and low-equivalence premise, STS:

- $\underline{t}_1 \sim \underline{t}_2 \implies \widehat{\text{obs}}(\lceil \underline{t}_1 \rceil) \approx_= \widehat{\text{obs}}(\lceil \underline{t}_2 \rceil)$

By Low-equivalence Soundness Element

□

Lemma B.1.1.12 (Low-equivalence Soundness Element)**.**

When projected, low-equivalent traces have equal probability distributions modulo adversary observation.

If: $\underline{t}_1 \sim \underline{t}_2$

Then: $\widehat{\text{obs}}(\lceil \underline{t}_1 \rceil) \approx_= \widehat{\text{obs}}(\lceil \underline{t}_2 \rceil)$

Proof. Induction on traces $\underline{t}_1$ and $\underline{t}_2$ and inversion on assumed low-equivalence

(1) Case $\underline{t}_1 = \underline{t}_2 = \epsilon$

Immediate

(2) Case $\underline{t}_1 = \underline{t}_1' \cdot \underline{\sigma}_1, \underline{e}_1$ and $\underline{t}_2 = \underline{t}_2' \cdot \underline{\sigma}_2, \underline{e}_2$

By inversion on assumed low-equivalence:

- $\underline{t'_1} \sim \underline{t'_2}$
- $\underline{\sigma}_1 \sim \underline{\sigma}_2$
- $\underline{e}_1 \sim \underline{e}_2$

By induction hypothesis:

- $\widehat{\text{obs}}(\lceil \underline{t'_1} \rceil) \approx_{=} \widehat{\text{obs}}(\lceil \underline{t'_2} \rceil)$

By Low-equivalence Soundness Element Store and Low-equivalence Soundness Element Expression:

- $\widehat{\text{obs}}(\lceil \underline{\sigma}_1 \rceil) \approx_{=} \widehat{\text{obs}}(\lceil \underline{\sigma}_2 \rceil)$
- $\widehat{\text{obs}}(\lceil \underline{e}_1 \rceil) \approx_{=} \widehat{\text{obs}}(\lceil \underline{e}_2 \rceil)$

Rewrite both sides by:

$$
\begin{aligned}
& \widehat{\text{obs}}(\lceil \underline{t'_i \cdot \varsigma_i} \rceil) \\
= \quad & \text{do} \ \ \overset{\bullet}{t} \leftarrow \widehat{\text{obs}}(\lceil \underline{t'_i} \rceil) \qquad \wr \ \text{defn. of } \lceil \_ \rceil \text{ and } \widehat{\text{obs}}, \text{ and Monad Laws} \ \int \\
& \quad \overset{\bullet}{\sigma} \leftarrow \widehat{\text{obs}}(\lceil \underline{\sigma}_i \rceil) \\
& \quad \overset{\bullet}{e} \leftarrow \widehat{\text{obs}}(\lceil \underline{e}_i \rceil) \\
& \quad \underline{\text{return}}(\overset{\bullet}{t} \cdot \overset{\bullet}{\sigma}, \overset{\bullet}{e})
\end{aligned}
$$

By iterated Bind Equivalence, Return Equivalence and three previously established facts

□

Lemma B.1.1.13 (Low-equivalence Soundness Element Store)**.**

When projected, low-equivalent stores have equal probability distributions modulo adversary observation.

If: $\underline{\sigma}_1 \sim \underline{\sigma}_2$

Then: $\widehat{\text{obs}}(\lceil \underline{\sigma}_1 \rceil) \approx_{=} \widehat{\text{obs}}(\lceil \underline{\sigma}_2 \rceil)$

Proof. Induction on $\underline{\sigma}_1$ and $\underline{\sigma}_2$, inversion on assumed low-equivalence, Monad Laws, Return Equivalence and Bind Equivalence. □

Lemma B.1.1.14 (Low-equivalence Soundness Element Expression)**.**

When projected, low-equivalent expressions have equal probability distributions modulo adversary observation.

If: $\underline{e}_1 \sim \underline{e}_2$

Then: $\widehat{\text{obs}}(\lceil \underline{e}_1 \rceil) \approx_= \widehat{\text{obs}}(\lceil \underline{e}_2 \rceil)$

Proof. Induction on $\underline{e}_1$ and $\underline{e}_2$, inversion on assumed low-equivalence, Monad Laws, Return Equivalence and Bind Equivalence. □

Simulation (Mixed)

Lemma B.1.1.15 (Simulation (Mixed))**.**

When projected, the mixed semantics simulates the intensional standard semantics on source expressions.

If: e *is a source expression*

Then: $\hat{\lceil}\underline{\text{nstep}}(N, \varnothing, e)\hat{\rceil} = \text{nstep}_{\mathcal{I}}(N, \varnothing, e)$

Proof. Induction on N

(1) Case $N = 0$:

$\lceil e \rceil = e$ (by Simulation (Mixed) Zero)

(2) Case $N = N + 1$:

$\hat{\lceil}\underline{\text{nstep}}(N, \varnothing, e)\hat{\rceil} = \text{nstep}_{\mathcal{I}}(N, \varnothing, e)$ *(IH)* (by inductive hypothesis)

By equational reasoning:

$\hat{\lceil}\underline{\text{nstep}}(N+1,\varnothing,e)\hat{\rceil}$

$=$ $\wr$ defn. of $\underline{\text{nstep}}$ and $\hat{\lceil}\_\hat{\rceil}$, Monad Laws and Monad Commutativity $\int$

do $\underline{t}\cdot\underline{\varsigma} \leftarrow \underline{\text{nstep}}(N,\varnothing,e)$

$t \leftarrow \lceil\underline{t}\rceil$

$\varsigma \leftarrow \lceil\underline{\varsigma}\rceil$

$\underline{\varsigma}' \leftarrow \underline{\text{step}}(N+1,\underline{\varsigma})$

$\varsigma' \leftarrow \lceil\underline{\varsigma}'\rceil$

$\text{return}(t\cdot\varsigma\cdot\varsigma')$

$=$ $\wr$ Simulation (Mixed) Single, Monad Laws, Monad Idempotence (Intensional Only) $\int$

do $\underline{t}\cdot\underline{\varsigma} \leftarrow \underline{\text{nstep}}(N,\varnothing,e)$

$t \leftarrow \lceil\underline{t}\rceil$

$\varsigma \leftarrow \lceil\underline{\varsigma}\rceil$

$\varsigma' \leftarrow \text{step}_{\mathcal{I}}(N+1,\varsigma)$

$\text{return}(t\cdot\varsigma\cdot\varsigma')$

$=$ $\wr$ *(IH)*, Monad Laws and defn. of $\lceil\_\rceil$ $\int$

do $t\cdot\varsigma \leftarrow \text{nstep}_{\mathcal{I}}(N,\varnothing,e)$

$\varsigma' \leftarrow \text{step}_{\mathcal{I}}(N+1,\varsigma)$

$\text{return}(t\cdot\varsigma\cdot\varsigma')$

$=$ $\wr$ defn. of $\text{nstep}_{\mathcal{I}}$ $\int$

$\text{nstep}_{\mathcal{I}}(N,\varnothing,e)$

□

Lemma B.1.1.16 (Simulation (Mixed) Zero)**.**

Projection on source expressions is the identity.

If: e *is a source expression*

Then: $\lceil e\rceil = e$

Proof. Induction on e

□

Lemma B.1.1.17 (Simulation (Mixed) Single)**.**

When projected, the mixed semantics simulates the intensional standard semantics on source expressions and on a single step.

$\hat{\lceil}\underline{\text{step}}(N, \underline{\sigma}, \underline{e})\hat{\rceil} = \widehat{\text{step}}_{\mathcal{I}}(N, \lceil\underline{\sigma}, \underline{e}\rceil)$

Proof. Induction on $\underline{e}$; first case is shown as representative trivial case; subsequent cases are non-trivial

- Case $\underline{e} = b_\ell$:

$$
\begin{aligned}
& \hat{\lceil}\underline{\text{step}}(N, \underline{\sigma}, b_\ell)\hat{\rceil} \\
= \quad & \lbag \text{ defn. of } \underline{\text{step}} \ \rbag \\
& \hat{\lceil}\text{return}(\underline{\sigma}, \texttt{bitv}_\ell(\text{return}(b)))\hat{\rceil} \\
= \quad & \lbag \text{ defn. of } \hat{\lceil}\_\hat{\rceil}, \text{step}_{\mathcal{I}} \text{ and Monad Laws } \rbag \\
& \text{do } \ \sigma \leftarrow \lceil\underline{\sigma}\rceil \\
& \quad\ e \leftarrow \lceil b_\ell\rceil \\
& \quad\ \text{step}_{\mathcal{I}}(\sigma, e) \\
= \quad & \lbag \text{ defn. of } \lceil\_\rceil, \widehat{\text{step}}_{\mathcal{I}} \text{ and Monad Laws } \rbag \\
& \widehat{\text{step}}_{\mathcal{I}}(N, \lceil\underline{\sigma}, b_\ell\rceil)
\end{aligned}
$$

- Case $\underline{e} = \texttt{flip}^\rho()$:

$$
\begin{aligned}
& \hat{\lceil}\underline{\text{step}}(N, \underline{\sigma}, \texttt{flip}^\rho())\hat{\rceil} \\
= \quad & \lbag \text{ defn. of } \underline{\text{step}} \ \rbag \\
& \hat{\lceil}\text{return}(\underline{\sigma}, \texttt{flipv}_\ell(\text{bit}(N)))\hat{\rceil} \\
= \quad & \lbag \text{ defn. of } \hat{\lceil}\_\hat{\rceil}, \text{step}_{\mathcal{I}} \text{ and Monad Laws } \rbag \\
& \text{do } b \leftarrow \text{bit}(N) \ ; \text{return}(\underline{\sigma}, \texttt{flipv}(b)) \\
= \quad & \lbag \text{ defn. of } \lceil\_\rceil, \widehat{\text{step}}_{\mathcal{I}} \text{ and Monad Laws } \rbag \\
& \widehat{\text{step}}_{\mathcal{I}}(N, \lceil\underline{\sigma}, \texttt{flip}^\rho()\rceil)
\end{aligned}
$$

- Case $\underline{e} = \texttt{cast}_P(\texttt{flipv}(\hat{b}))$:

$$
\begin{aligned}
& \hat{\lceil}\underline{\mathrm{step}}(N, \underline{\sigma}, \mathtt{cast}_P(\mathtt{flipv}(\hat{b})))\hat{\rceil} \\
= \quad & \text{⦇ defn. of } \underline{\mathrm{step}} \text{ ⦈} \\
& \hat{\lceil}\mathrm{do}\ b \leftarrow \hat{b}\ ;\ \mathrm{return}(\underline{\sigma}, \mathtt{bitv}_P(\mathrm{return}(b)))\hat{\rceil} \\
= \quad & \text{⦇ defn. of } \hat{\lceil}\_\hat{\rceil}, \mathrm{step}_{\mathcal{I}} \text{ and Monad Laws ⦈} \\
& \mathrm{do}\ b \leftarrow \hat{b}\ ;\ \mathrm{return}(\underline{\sigma}, \mathtt{bitv}_P(b)) \\
= \quad & \text{⦇ defn. of } \lceil\_\rceil, \widehat{\mathrm{step}}_{\mathcal{I}} \text{ and Monad Laws ⦈} \\
& \widehat{\mathrm{step}}_{\mathcal{I}}(N, \lceil\underline{\sigma}, \mathtt{cast}_P(\mathtt{flipv}(\hat{b}))\rceil)
\end{aligned}
$$

- *All other cases are analogous to above cases.*

□

Simulation (Intensional)

Lemma B.1.1.18 (Simulation (Intensional))**.**

The intensional standard semantics simulates the ground truth semantics on source expressions.

$$\Pr\left[\mathrm{nstep}_{\mathcal{D}}(N, \varnothing, e) \doteq \varsigma\right] = \Pr\left[\mathrm{nstep}_{\mathcal{I}}(N, \varnothing, e) \doteq \varsigma\right]$$

Proof. Induction on N and by Bind Probability, Return Probability and Simulation (Intensional) □

Lemma B.1.1.19 (Simulation (Intensional) Single)**.**

The intensional standard semantics simulates the ground truth semantics on source expressions, and on a single step.

$$\Pr\left[\mathrm{step}_{\mathcal{D}}(N, \sigma, e) \doteq \varsigma\right] = \Pr\left[\mathrm{step}_{\mathcal{I}}(N, \sigma, e) \doteq \varsigma\right]$$

Proof. Induction on e and by Bind Probability, Return Probability and $\mathrm{bit}_{\mathcal{I}}(N{+}1) \perp\!\!\!\perp \mathrm{nstep}_{\mathcal{I}}(N, \varsigma)$, which is true by $\mathrm{height}(\mathrm{nstep}(N, \varsigma)) \leq N$ and $\mathrm{bit}_{\mathcal{I}}(N+1) \perp\!\!\!\perp \hat{x}$ when $\mathrm{height}(\hat{x}) \leq N$ □

B.1.2 Type Preservation

Lemma B.1.1.20 (Type Preservation)**.**

Well-typing is preserved by the mixed semantics w.r.t. new trace history.

If: e *is a closed source expression*

And: $\vdash e : \tau$

And: $\underline{t \cdot \varsigma} \in \mathrm{support}(\underline{\mathrm{nstep}}(N, \varnothing, e))$

Let: $\Phi \triangleq [\underline{\mathrm{nstep}}(N, \varnothing, e) \doteq \underline{t \cdot \varsigma}]$

Then: there exists Σ *and* Ψ

S.t.: $\Phi, \Sigma \vdash \underline{\varsigma} : \tau, \Psi$

Proof. By Type Preservation (Strong) which has a stronger conclusion (and therefore induction hypothesis)

□

Lemma B.1.1.21 (Type Preservation (Strong))**.**

Well-typing is preserved by the mixed semantics w.r.t. new trace history.

If: e *is a closed source expression*

And: $\vdash e : \tau$

And: $\underline{t}{\cdot}\underline{\varsigma} \in \mathrm{support}(\underline{\mathrm{nstep}}(N, \varnothing, e))$

Let: $\Phi \triangleq [\underline{\mathrm{nstep}}(N, \varnothing, e) \doteq \underline{t}{\cdot}\underline{\varsigma}]$

Then: there exists Σ *and* Ψ

S.t.: $\Phi, \Sigma \vdash \underline{\varsigma} : \tau, \Psi$

And: $\forall \hat{b} \in \Psi.\ \hat{b} \perp\!\!\!\perp \{\mathrm{bit}(N') \mid N' \geq N + 1\}$

Proof. Induction on N

(1) Case $N = 0$:

$\Phi = [\underline{\mathrm{nstep}}(0, \varnothing, e) \doteq \mathrm{return}(\varnothing, e)] = [\mathit{true}]$

$\Sigma = \varnothing$

$\Psi = \varnothing$

$\varnothing, \varnothing, \varnothing \vdash \varnothing, e : \tau, \varnothing$ (by Source Expression Mixed Typing)

(2) Case $N = N + 1$:

By induction hypothesis *(IH)*:

- $\Phi', \Sigma' \vdash \underline{\varsigma}' : \tau, \Psi'$ for some Σ', Ψ'

  and where $\Phi' \triangleq [\underline{\mathrm{nstep}}(N, \varnothing, e) \doteq \underline{t}'{\cdot}\underline{\varsigma}']$

  and where $\underline{t} = \underline{t}'{\cdot}\underline{\varsigma}'$

- $\forall \hat{b} \in \Psi'.\ \hat{b} \perp\!\!\!\perp \{\mathrm{bit}(N') \mid N' \geq N + 1\}$

By Type Preservation Single and second fact due to *(IH)*:

- $\Phi'', \Sigma'' \vdash \underline{\varsigma} : \tau, \Psi''$ for some Σ'', Ψ''

and where $\Phi'' \triangleq [\Phi', \underline{\mathrm{step}}(N, \underline{\varsigma}') \doteq \underline{\varsigma}]$

- $\forall \hat{b} \in \Psi''.\ \hat{b} \perp\!\!\!\perp \{\mathrm{bit}(N') \mid N' \geq N + 1 + 1\}$

Construct $\Sigma \triangleq \Sigma''$ and $\Psi \triangleq \Psi''$; by previous typing and bit independence, and $\Phi'' = \Phi$ (b.c. $\underline{t} = \underline{t}' \cdot \underline{\varsigma}'$)

□

Lemma B.1.1.22 (Source Expression Mixed Typing)**.**

Well-typed source expressions are well-typed in the mixed type system.

If: e *is a source expression*

And: $\vdash e : \tau$ *(via source expression typing)*

Then: $\vdash e : \tau$ *(via mixed evaluation typing)*

Proof. Induction on e and inversion on assumed well-typing □

Lemma B.1.1.23 (Type Preservation Single)**.**

Well-typing is preserved by the mixed semantics w.r.t. new trace history on a single step.

If: $\Phi, \Sigma \vdash \underline{\varsigma} : \tau, \Psi$

And: $\forall \hat{b} \in \Psi.\ \hat{b} \perp\!\!\!\perp \{\mathrm{bit}(N') \mid N' \geq N\}$

And: $\underline{\varsigma}' \in \mathrm{support}(\mathrm{step}_{\mathcal{I}}(N, \underline{\varsigma}))$

Let: $\Phi' \triangleq [\Phi, \mathrm{step}_{\mathcal{I}}(N, \underline{\varsigma}) \doteq \underline{\varsigma}']$

Then: there exists Σ' *and* Ψ'

S.t.: $\Phi', \Sigma' \vdash \underline{\varsigma}' : \tau, \Psi'$

And: $\forall \hat{b} \in \Psi'.\ \hat{b} \perp\!\!\!\perp \{\mathrm{bit}(N') \mid N' \geq N + 1\}$

Proof. By Progress (Mixed) and definition of $\mathrm{step}_{\mathcal{I}}$; two cases:

(1) $\underline{\varsigma} = \underline{\sigma}, \underline{v}$

$\underline{\varsigma}' = \underline{\varsigma}\,,\ \Phi' = \Phi\,,\ \Sigma' = \Sigma$ and $\Psi' = \Psi$

Immediate

(2) $\underline{\varsigma} = \underline{\sigma}, \underline{E}[\underline{e}]$

$\underline{\varsigma}' = \underline{\sigma}', \underline{E}[\underline{e}']$ for $\underline{\sigma}', \underline{e}' \in \mathrm{support}(\mathrm{step}_{\mathcal{I}}(N, \underline{\sigma}, \underline{e}))$

By Contexts Preserve Typing:

- There exists τ', Ψ_c and Ψ'

  S.t.: $\Psi_c, \Phi, \Sigma \vdash \underline{\sigma}, \underline{e} : \tau' \,;\, \Psi'$

  And: $\Psi_c \uplus \Psi' = \Psi$

By Type Preservation Redex:

- There exists Σ' and Ψ''

  S.t.: $\Sigma' \supseteq \Sigma$

  And: $\Psi_c, \Phi', \Sigma' \vdash \underline{\sigma}', \underline{e}' : \tau' \,;\, \Psi''$

  And: $\forall \rho, \hat{b}.\ \Psi_c \setminus (\{\hat{b}\}, \varnothing) \uplus \Psi', \Phi \vdash \hat{b} : \mathtt{flip}^{\rho} \implies \Psi_c \setminus (\{\hat{b}\}, \varnothing) \uplus \Psi'', \Phi' \vdash \hat{b} : \mathtt{flip}^{\rho}$

  And: $\forall \hat{b} \in \Psi_c \uplus \Psi''.\ \hat{b} \perp\!\!\!\perp \{\mathrm{bit}(N') \mid N' \geq N + 1\}$

By Weaken Context and Contexts Preserve Typing:

- $\Phi', \Sigma' \vdash \underline{\sigma}', \underline{E}[\underline{e}'] : \tau \,;\, \Psi_c \uplus \Psi''$

Construct $\Sigma' \triangleq \Sigma'$ and $\Psi' = \Psi_c \uplus \Psi''$; by previous typing and bit independence

□

Lemma B.1.1.24 (Contexts Preserve Typing)**.**

If: $\varnothing, \Phi, \Sigma, \varnothing \vdash \underline{E}[\underline{e}] : \tau \,;\, \varnothing, \Psi$

Then: there exists τ', Ψ_c, Ψ' *s.t.:*

(1) $\Psi_c, \Phi, \Sigma, \varnothing \vdash \underline{e} : \tau' \,;\, \varnothing, \Psi'$ *and* $\Psi_c \uplus \Psi' = \Psi$

(2) $\Psi_c, \Phi, \Sigma, \varnothing \vdash \underline{e}' : \tau' \,;\, \varnothing, \Psi'$ *and* $\Psi_c \uplus \Psi' = \Psi \implies \Psi_c, \Phi, \Sigma, \varnothing \vdash \underline{E}[\underline{e}'] : \tau \,;\, \varnothing, \Psi$

Proof. Induction on E and inversion on $\Psi_c, \Phi, \Sigma, \varnothing \vdash \underline{E[e]} : \tau \; ; \varnothing, \Psi$

□

Type Preservation Redex

Lemma B.1.1.25 (Type Preservation Redex)**.**

If: $\underline{e}$ *a redex*

And: $\Psi_c, \Phi, \Sigma \vdash \underline{\sigma}, \underline{e} : \tau \; ; \Psi$

And: $\forall \hat{b} \in \Psi_c \uplus \Psi.\ \hat{b} \perp\!\!\!\perp \{\mathrm{bit}(N') \mid N' \geq N\}$

And: $\underline{\varsigma} \in \mathrm{support}(\mathrm{step}_{\mathcal{I}}(N, \underline{\sigma}, \underline{e}))$

Let: $\Phi' \triangleq [\Phi, \mathrm{step}_{\mathcal{I}}(N, \underline{\sigma}, \underline{e}) \doteq \underline{\varsigma}]$

Then: there exists Σ' *and* Ψ'

S.t.: $\Sigma' \supseteq \Sigma$

And: $\Psi_c, \Phi', \Sigma' \vdash \underline{\varsigma} : \tau \; ; \Psi'$

And: $\forall \rho, \hat{b}.\ \Psi_c \setminus (\{\hat{b}\}, \varnothing) \uplus \Psi, \Phi \vdash \hat{b} : \texttt{flip}^{\rho} \implies \Psi_c \setminus (\{\hat{b}\}, \varnothing) \uplus \Psi', \Phi' \vdash \hat{b} : \texttt{flip}^{\rho}$

And: $\forall \hat{b} \in \Psi_c \uplus \Psi'.\ \hat{b} \perp\!\!\!\perp \{\mathrm{bit}(N') \mid N' \geq N + 1\}$

Proof. By inversion:

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

Case analysis on $\underline{e}$:

(1) $\underline{e} = \texttt{flip}^{\rho}()$

By inversion:

$$\frac{}{\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma, \varnothing \vdash \texttt{flip}^\rho() : \texttt{flip}^\rho \; ; \oslash, \varnothing, \varnothing}$$

$\tau = \texttt{flip}^\rho$

$\Psi_e = \varnothing, \varnothing$

$\underline{\varsigma} = \underline{\sigma}, \texttt{flipv}(\mathrm{bit}(N))$

$\Phi' = [\Phi, \mathrm{step}_{\mathcal{I}}(N, \underline{\sigma}, \texttt{flip}^\rho() \doteq \underline{\sigma}, \texttt{flipv}(\mathrm{bit}(N))] = \Phi$

$\Psi'_e \triangleq \{\mathrm{bit}(N)\}, \varnothing$

Construct $\Sigma' \triangleq \Sigma \supseteq \Sigma$

Construct $\Psi' \triangleq \Psi_\sigma \uplus (\{\mathrm{bit}(N)\}, \varnothing) = \Psi_\sigma \uplus \Psi'_e$

To show:

(a) $\Psi_c \uplus \{\mathrm{bit}(N)\}, \Phi, \Sigma \vdash \underline{\sigma} \; ; \Psi_\sigma$

(b) $\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma \vdash \texttt{flipv}(\mathrm{bit}(N)) : \texttt{flip}^\rho \; ; \varnothing$

(c) $\forall \rho', \hat{b}'.$

$\Psi_c \setminus (\{\hat{b}'\}, \varnothing) \uplus \Psi_\sigma, \Phi, \Sigma \vdash \hat{b}' : \texttt{flip}^{\rho'}$

$\Longrightarrow$

$\Psi_c \setminus (\{\hat{b}'\}, \varnothing) \uplus \Psi_\sigma \uplus (\{\mathrm{bit}(N)\}, \varnothing), \Phi, \Sigma \vdash \hat{b}' : \texttt{flip}^{\rho'}$

(d) $\forall \hat{b} \in \Psi_c \uplus \Psi_\sigma \uplus (\{\mathrm{bit}(N)\}, \varnothing).\ \hat{b} \perp\!\!\!\perp \{\mathrm{bit}(N') \mid N' \geq N + 1\}$

(a) is by Weaken Store and Type Preservation: Flip applied to Ψ_c

(b-c) are by Type Preservation: Flip applied to $\Psi_c \uplus \Psi_\sigma$

(d) is by assumed bit independence and Bit Independence

(2) $\underline{e} = \texttt{cast}_P(\texttt{flipv}(\hat{b}))$

By inversion:

$$\frac{\dfrac{\Psi_c \uplus \Psi_\sigma, \Phi \vdash \hat{b} : \texttt{flip}^\rho}{\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma \vdash \texttt{flipv}(\hat{b}) : \texttt{flip}^\rho \; ; \{\hat{b}\}, \varnothing}}{\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma, \varnothing \vdash \texttt{cast}_P(\texttt{flipv}(\hat{b})) : \texttt{bit}^\perp_P \; ; \oslash, \{\hat{b}\}, \varnothing}$$

$\tau = \texttt{bit}^\perp_P$

$\Psi_e = \{\hat{b}\}, \varnothing$

$\underline{\varsigma} = \underline{\sigma}, \mathtt{bitv}_P(\mathrm{return}(b))$ for $b \in \{\mathtt{0}, \mathtt{1}\}$

$\Phi' = [\Phi, \mathrm{step}_{\mathcal{I}}(N, \underline{\sigma}, \mathtt{cast}_P(\mathtt{flipv}(\hat{b}))) \doteq \underline{\sigma}, \mathtt{bitv}_P(\mathrm{return}(b))] = [\Phi, \hat{b} \doteq b]$

$\Psi'_e \triangleq \varnothing$

Construct $\Sigma' \triangleq \Sigma \supseteq \Sigma$

Construct $\Psi' \triangleq \Psi_\sigma = \Psi_\sigma \uplus \Psi'_e$

To show:

(a) $\Psi_c, [\Phi, \hat{b} \doteq b], \Sigma \vdash \underline{\sigma} : \Psi_\sigma$

(b) $\Psi_c \uplus \Psi_\sigma, [\Phi, \hat{b} \doteq b], \Sigma \vdash \mathtt{bitv}_P(\mathrm{return}(b)) : \mathtt{bit}_P^{\perp} ; \varnothing$

(c) $\forall \rho', \hat{b}'.$

$\Psi_c \setminus (\{\hat{b}'\}, \varnothing) \uplus \Psi_\sigma \uplus (\{\hat{b}\}, \varnothing), \Phi, \Sigma \vdash \hat{b}' : \mathtt{flip}^{\rho'}$

$\Longrightarrow$

$\Psi_c \setminus (\{\hat{b}'\}, \varnothing) \uplus \Psi_\sigma, [\Phi, \hat{b} \doteq b], \Sigma \vdash \hat{b}' : \mathtt{flip}^{\rho'}$

(d) $\forall \hat{b} \in \Psi_c \uplus \Psi_\sigma.\ \hat{b} \perp\!\!\!\perp \{\mathrm{bit}(N') \mid N' \geq N + 1\}$

(a) is by Weaken Store and Type Preservation: CastP applied to Ψ_c

(b-c) are by Type Preservation: CastP applied to $\Psi_c \uplus \Psi_\sigma$

(d) is by assumption

(3) $\underline{e} = \mathtt{cast}_S(\mathtt{flipv}(\hat{b}))$

By inversion:

$$\dfrac{\dfrac{\Psi_c \uplus \Psi_\sigma, \Phi \vdash \hat{b} : \mathtt{flip}^{\rho}}{\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma \vdash \mathtt{flipv}(\hat{b}) : \mathtt{flip}^{\rho} ; \varnothing, \{\rho \mapsto \hat{b}\}}}{\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma, \varnothing \vdash \mathtt{cast}_S(\mathtt{flipv}(\hat{b})) : \mathtt{bit}_S^{\rho} ; \oslash, \varnothing, \{\rho \mapsto \{\hat{b}\}\}}$$

$\tau = \mathtt{bit}_S^{\rho}$

$\Psi_e = \varnothing, \{\rho \mapsto \{\hat{b}\}\}$

$\underline{\varsigma} = \underline{\sigma}, \mathtt{bitv}_S(\hat{b})$

$\Phi' = [\Phi, \mathrm{step}_{\mathcal{I}}(N, \underline{\sigma}, \mathtt{cast}_S(\mathtt{flipv}(\hat{b}))) \doteq \underline{\sigma}, \mathtt{bitv}_S(\hat{b})] = \Phi$

$\Psi'_e \triangleq \varnothing, \{\rho \mapsto \{\hat{b}\}\} = \Psi_e$

$\Sigma \triangleq \Sigma \supseteq \Sigma$

$\Psi' \triangleq \Psi_\sigma \uplus \Psi_e = \Psi_\sigma \uplus \Psi'_e$

To show:

(a) $\Psi_c \uplus (\varnothing, \{\rho \mapsto \{\hat{b}\}\}), \Phi, \Sigma \vdash \underline{\sigma} ; \Psi_\sigma$

(b) $\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma \vdash \mathtt{bitv}_S(\hat{b}) : \mathtt{bit}_S^\rho ; \varnothing, \{\rho \mapsto \{\hat{b}\}\}$

(c) $\forall \rho', \hat{b}'.$

$\Psi_c \setminus (\{\hat{b}'\}, \varnothing) \uplus \Psi_\sigma \uplus (\varnothing, \{\rho \mapsto \{\hat{b}\}\}), \Phi, \Sigma \vdash \hat{b}' : \mathtt{flip}^{\rho'}$

$\implies$

$\Psi_c \setminus (\{\hat{b}'\}, \varnothing) \uplus \Psi_\sigma \uplus (\varnothing, \{\rho \mapsto \{\hat{b}\}\}), \Phi, \Sigma \vdash \hat{b}' : \mathtt{flip}^{\rho'}$

(d) $\forall \hat{b} \in \Psi_c \uplus \Psi_\sigma \uplus (\varnothing, \{\rho \mapsto \{\hat{b}\}\}).\ \hat{b} \perp\!\!\!\perp \{\mathrm{bit}(N') \mid N' \geq N + 1\}$

(a) is by assumption

(b) is immediate

(c) is immediate

(d) is by assumption

(4) $\underline{e} = \mathtt{mux}(\mathtt{bitv}_S(\hat{b}_1), \mathtt{bitv}_S(\hat{b}_2), \mathtt{bitv}_S(\hat{b}_3))$

By inversion:

$$\dfrac{\dfrac{}{\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma \vdash \mathtt{bitv}_S(\hat{b}_1) : \mathtt{bit}_S^{\rho_1} ; \varnothing, \{\rho_1 \mapsto \{\hat{b}_1\}\}} \quad \dfrac{}{\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma \vdash \mathtt{bitv}_S(\hat{b}_2) : \mathtt{bit}_S^{\rho_2} ; \varnothing, \{\rho_2 \mapsto \{\hat{b}_2\}\}} \quad \dfrac{}{\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma \vdash \mathtt{bitv}_S(\hat{b}_3) : \mathtt{bit}_S^{\rho_3} ; \varnothing, \{\rho_3 \mapsto \{\hat{b}_3\}\}}}{\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma, \varnothing \vdash \mathtt{mux}(\mathtt{bitv}_S(\hat{b}_1), \mathtt{bitv}_S(\hat{b}_2), \mathtt{bitv}_S(\hat{b}_3)) : \mathtt{bit}_S^{\rho_1 \sqcup \rho_2 \sqcup \rho_3} \times \mathtt{bit}_S^{\rho_1 \sqcup \rho_2 \sqcup \rho_3} ; \oslash, \varnothing, \{\rho_i \mapsto \{\hat{b}_i\}\}}$$

$\tau = \mathtt{bit}_S^{\rho_1 \sqcup \rho_2 \sqcup \rho_3} \times \mathtt{bit}_S^{\rho_1 \sqcup \rho_2 \sqcup \rho_3}$

$\Psi_e = \varnothing, \{\rho_i \mapsto \{\hat{b}_i\}\}$

$\underline{\varsigma} = \underline{\sigma}, \langle \mathtt{bitv}_S(\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)), \mathtt{bitv}_S(\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)) \rangle$

$\Phi' = \left[\Phi, \mathrm{step}_{\mathcal{I}}(N, \underline{\sigma}, \mathtt{mux}(\mathtt{bitv}_S(\hat{b}_1), \mathtt{bitv}_S(\hat{b}_2), \mathtt{bitv}_S(\hat{b}_3))) \doteq \underline{\sigma}, \langle \mathtt{bitv}_S(\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)), \mathtt{bitv}_S(\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2))\rangle\right] =$

Φ

$\Psi'_e = \varnothing, \{\rho_1 \sqcup \rho_2 \sqcup \rho_3 \mapsto \{\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3), \widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)\}\}$

$\Sigma \triangleq \Sigma \supseteq \Sigma$

$\Psi' \triangleq \Psi_\sigma \uplus (\varnothing, \{\rho_1 \sqcup \rho_2 \sqcup \rho_3 \mapsto \{\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3), \widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)\}\}) = \Psi_\sigma \uplus \Psi'_e$

To show:

(a) $\Psi_c \uplus \Psi'_e, \Phi, \Sigma \vdash \underline{\sigma} : \Psi_\sigma$

(b) $\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma \vdash \langle \mathtt{bitv}_S(\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)), \mathtt{bitv}_S(\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2))\rangle : \mathtt{bit}_S^{\rho_1 \sqcup \rho_2 \sqcup \rho_3} \times \mathtt{bit}_S^{\rho_1 \sqcup \rho_2 \sqcup \rho_3}$; Ψ'_e

(c) $\forall \rho', \hat{b}'.$

$\Psi_c \setminus (\{\hat{b}'\}, \varnothing) \uplus \Psi_\sigma \uplus \Psi_e, \Phi, \Sigma \vdash \hat{b}' : \mathtt{flip}^{\rho'}$

$\Longrightarrow$

$\Psi_c \setminus (\{\hat{b}'\}, \varnothing) \uplus \Psi_\sigma \uplus \Psi'_e, \Phi, \Sigma \vdash \hat{b}' : \mathtt{flip}^{\rho'}$

(d) $\forall \hat{b} \in \Psi_c \uplus \Psi_\sigma \uplus \Psi'_e. \{\hat{b} \perp\!\!\!\perp \mathrm{bit}(N') \mid N' \geq N + 1\}$

(a) is by assumption

(b) is immediate

(c) is by Type Preservation: Mux BitS applied to $\Psi_c \uplus \Psi_\sigma$

(d) is by assumption and Cond Independence

(5) $\underline{e} = \mathtt{mux}(\mathtt{bitv}_S(\hat{b}_1), \mathtt{flipv}(\hat{b}_2), \mathtt{flipv}(\hat{b}_3))$

By inversion:

$$\dfrac{\dfrac{}{\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma \vdash \mathtt{bitv}_S(\hat{b}_1) : \mathtt{bit}_S^{\rho_1} ; \varnothing, \{\rho_1 \mapsto \{\hat{b}_1\}\}} \quad \rho_1 \sqsubset \rho_2 \quad \rho_1 \sqsubset \rho_3 \qquad \dfrac{\Psi_c \uplus \Psi_\sigma, \Phi \vdash \hat{b}_2 : \mathtt{flip}^{\rho_2}}{\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma \vdash \mathtt{flipv}(\hat{b}_2) : \mathtt{flip}^{\rho_2} ; \{\hat{b}_2\}, \varnothing} \qquad \dfrac{\Psi_c \uplus \Psi_\sigma, \Phi \vdash \hat{b}_3 : \mathtt{flip}^{\rho_3}}{\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma \vdash \mathtt{flipv}(\hat{b}_3) : \mathtt{flip}^{\rho_3} ; \{\hat{b}_3\}, \varnothing}}{\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma, \varnothing \vdash \mathtt{mux}(\mathtt{bitv}_S(\hat{b}_1), \mathtt{flipv}(\hat{b}_2), \mathtt{flipv}(\hat{b}_3)) : \mathtt{flip}^{\rho_2 \sqcap \rho_3} \times \mathtt{flip}^{\rho_2 \sqcap \rho_3} ; \varnothing, \{\hat{b}_2, \hat{b}_3\}, \{\rho_1 \mapsto \{\hat{b}_1\}\}}$$

$\tau = \mathtt{flip}^{\rho_2 \sqcap \rho_3} \times \mathtt{flip}^{\rho_2 \sqcap \rho_3}$

$\Psi_e = \{\hat{b}_2, \hat{b}_3\}, \{\rho_1 \mapsto \{\hat{b}_1\}\}$

$\underline{\varsigma} = \underline{\sigma}, \langle \texttt{flipv}(\widehat{\text{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)), \texttt{flipv}(\widehat{\text{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)) \rangle$

$\Phi' = [\Phi, \text{step}_{\mathcal{I}}(N, \underline{\sigma}, \texttt{mux}(\texttt{bitv}_S(\hat{b}_1), \texttt{flipv}(\hat{b}_2), \texttt{flipv}(\hat{b}_3))) \doteq \underline{\sigma}, \langle \texttt{flipv}(\widehat{\text{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)), \texttt{flipv}(\widehat{\text{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)) \rangle] =$

Φ

$\Psi'_e = \{\widehat{\text{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3), \widehat{\text{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)\}, \varnothing$

$\Sigma \triangleq \Sigma \supseteq \Sigma$

$\Psi' \triangleq \Psi_\sigma \uplus (\{\widehat{\text{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3), \widehat{\text{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)\}, \varnothing) = \Psi_\sigma \uplus \Psi'_e$

To show:

(a) $\Psi_c \uplus \Psi'_e, \Phi, \Sigma \vdash \underline{\sigma} : \Psi_\sigma$

(b) $\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma \vdash \langle \texttt{flipv}(\widehat{\text{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)), \texttt{flipv}(\widehat{\text{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)) \rangle : \texttt{flip}^{\rho_2 \sqcap \rho_3} \times \texttt{flip}^{\rho_2 \sqcap \rho_3} ; \Psi'_e$

(c) $\forall \rho', \hat{b}'.$

$\Psi_c \setminus (\{\hat{b}'\}, \varnothing) \uplus \Psi_\sigma \uplus \Psi_e, \Phi, \Sigma \vdash \hat{b}' : \texttt{flip}^{\rho'}$

$\Longrightarrow$

$\Psi_c \setminus (\{\hat{b}'\}, \varnothing) \uplus \Psi_\sigma \uplus \Psi'_e, \Phi, \Sigma \vdash \hat{b}' : \texttt{flip}^{\rho'}$

(d) $\forall \hat{b} \in \Psi_c \uplus \Psi_\sigma \uplus \Psi'_e. \{\hat{b} \perp\!\!\!\perp \text{bit}(N') \mid N' \geq N + 1\}$

(a) is by Weaken Store and Type Preservation: Flip applied to Ψ_c

(b-c) are by Type Preservation: Flip applied to $\Psi_c \uplus \Psi_\sigma$

(d) is by assumption and Cond Independence

(6) $\underline{e} = \texttt{xor}(\texttt{bitv}_S(\hat{b}_1), \texttt{flipv}(\hat{b}_2))$

Analogous to mux-flip case

(7) $\underline{e} = \texttt{let } x = \underline{v} \texttt{ in } \underline{e}$

By inversion:

$$\frac{\Psi_c \uplus \Psi_\sigma \uplus \Psi_e, \Phi, \Sigma, \varnothing \vdash \underline{v} : \tau' ; \varnothing, \Psi_v \qquad \Psi_c \uplus \Psi_\sigma \uplus \Psi_v, \Phi, \Sigma, [x \mapsto \tau'] \vdash \underline{e}' : \tau ; \_, \Psi'_e}{\Psi_c \uplus \Psi_\sigma, \Phi, \Sigma, \varnothing \vdash \texttt{let } x = \underline{v} \texttt{ in } \underline{e}' : \tau ; \varnothing, \Psi_v \uplus \Psi'_e}$$

$\Psi_e = \Psi_v \uplus \Psi'_e$

$\underline{\varsigma} = \underline{\sigma}, [\underline{v}/x]\underline{e}$

$\Phi' = [\Phi, \mathrm{step}_{\mathcal{I}}(N, \underline{\sigma}, \texttt{let}\ x = \underline{v}\ \texttt{in}\ \underline{e}')) \doteq \underline{\sigma}, [\underline{v}/x]\underline{e}] = \Phi$

By Type Preservation: Substitution:

- There exists Ψ'_v

  S.t.: $\Psi'_v \subseteq \Psi_v$

  And: $\Psi_c, \Phi, \Sigma, \Gamma \vdash [\underline{v}/x]\underline{e} : \tau_2 \ ; \ \Gamma', \Psi'_v \uplus \Psi'_e$

$\Sigma \triangleq \Sigma \supseteq \Sigma$

$\Psi' \triangleq \Psi_\sigma \uplus \Psi'_v \uplus \Psi'_e$

To show:

(a) $\Psi_c \uplus \Psi'_v \uplus \Psi'_e, \Phi, \Sigma \vdash \underline{\sigma} \ ; \ \Psi_\sigma$

(b) $\Psi_c, \Phi, \Sigma, \Gamma \vdash [\underline{v}/x]\underline{e} : \tau_2 \ ; \ \Gamma', \Psi'_v \uplus \Psi'_e$

(c) $\forall \rho', \hat{b}'.$

$\Psi_c \setminus (\{\hat{b}'\}, \varnothing) \uplus \Psi_\sigma \uplus \Psi_v \uplus \Psi'_e, \Phi, \Sigma \vdash \hat{b}' : \texttt{flip}^{\rho'}$

$\implies$

$\Psi_c \setminus (\{\hat{b}'\}, \varnothing) \uplus \Psi_\sigma \uplus \Psi'_v \uplus \Psi'_e, \Phi, \Sigma \vdash \hat{b}' : \texttt{flip}^{\rho'}$

(d) $\forall \hat{b} \in \Psi_c \uplus \Psi_\sigma \uplus \Psi'_v \uplus \Psi'_e. \ \{\hat{b} \perp\!\!\!\perp \mathrm{bit}(N') \mid N' \geq N + 1\}$

(a) is by Weaken Store and Weaken Flip

(b) is by Type Preservation: Substitution

(c) is by Weaken Flip

(d) is by assumed bit independence

(8) $\underline{e} = \texttt{let}\ x, y = \underline{v}\ \texttt{in}\ \underline{e}$ and $\underline{e} = (\texttt{fun}_y(x : \tau).\ \underline{e})(\underline{v})$

Analogous to single-variable let-binding case

□

Lemma B.1.1.26 (Type Preservation: Flip)**.**

If: $\forall \hat{b} \in \Psi^F, \Psi^B.\ \hat{b} \perp\!\!\!\perp \{\mathrm{bit}(N') \mid N' \geq N\}$

Then: $\forall \rho', \hat{b}'.\ \Psi^F \setminus \{\hat{b}\}, \Psi^B, \Phi \vdash \hat{b}' : \mathtt{flip}^{\rho'} \implies \Psi \setminus \{\hat{b}\} \uplus \{\mathrm{bit}(N)\}, \Psi^B, \Phi \vdash \hat{b}' : \mathtt{flip}^{\rho'}$

Proof. Assume some ρ', $\hat{b}'$ where $\Psi^F \setminus \{\hat{b}'\}, \Psi^B, \Phi \vdash \hat{b}' : \mathtt{flip}^{\rho'}$

By inversion:

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

STS:

- $\left[\hat{b}' \perp\!\!\!\perp \mathrm{bit}(N), \Psi^F \setminus \{\hat{b}'\}, \Psi^B(\{\rho'' \mid \rho'' \sqsubset \rho'\}) \;\middle|\; \Phi\right]$

By assumption of bit independence and second inversion fact

□

Lemma B.1.1.27 (Type Preservation: CastP)**.**

If: $\Psi^F, \Psi^B, \Phi \vdash \hat{b} : \mathtt{flip}^{\rho}$

Then:

(1) $\Psi_c, [\Phi, \hat{b} \doteq b] \vdash \mathrm{return}(b) : \mathtt{bit}^{\perp}_{P}\ ;\ \varnothing$

(2) $\forall \rho', \hat{b}'.\ \Psi^F \setminus \{\hat{b}'\} \uplus \{\hat{b}\}, \Psi^B, \Phi \vdash \hat{b}' : \mathtt{flip}^{\rho'} \implies \Psi^F \setminus \{\hat{b}'\}, \Psi^B, [\Phi, \hat{b} \doteq b] \vdash \hat{b}' : \mathtt{flip}^{\rho'}$

Proof.(1) Immediate by constructing type derivation

(2) Assume some ρ' and $\hat{b}'$ where $\Psi^F \setminus \{\hat{b}'\} \uplus \{\hat{b}\}, \Psi^B, \Phi \vdash \hat{b}' : \mathtt{flip}^{\rho'}$

By inversion:

- $\Pr\left[\hat{b}' \doteq \mathtt{I} \;\middle|\; \Phi\right] = 1/2$ *(H1)*
- $\left[\hat{b}' \perp\!\!\!\perp \hat{b}, \Psi^F \setminus \{\hat{b}'\}, \Psi^B(\{\rho'' \mid \rho'' \sqsubset \rho'\}) \;\middle|\; \Phi\right]$ *(H2)*

STS:

(a) $\Pr\left[\hat{b}' \doteq \mathtt{I} \;\middle|\; \Phi, \hat{b} \doteq b\right] = 1/2$

(b) $\left[\hat{b}' \perp\!\!\!\perp \Psi^F \setminus \{\hat{b}'\}, \Psi^B(\{\rho'' \mid \rho'' \sqsubset \rho'\}) \;\middle|\; \Phi, \hat{b} \doteq b\right]$

(a) is by Decomposition applied *(H2)* to establish $\left[\hat{b}' \perp\!\!\!\perp \hat{b} \;\middle|\; \Phi\right]$, which is then applied to *(H1)*

(b) is by Decomposition applied to *(H2)*, moving $\hat{b}$ from the RHS of independence into the condition

□

Lemma B.1.1.28 (Type Preservation: Mux BitS)**.**

If: $\Psi^F \setminus \{\hat{b}'\}, \Psi^B \cup \{\rho_1 \mapsto \{\hat{b}_1\}, \rho_2 \mapsto \{\hat{b}_2\}, \rho_3 \mapsto \{\hat{b}_3\}\}, \Phi \vdash \hat{b}' : \mathtt{flip}^{\rho'}$

Then: $\Psi^F \setminus \{\hat{b}'\}, \Psi^B \cup \{\rho_1 \sqcup \rho_2 \sqcup \rho_3 \mapsto \{\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)\}\}, \Phi \vdash \hat{b}' : \mathtt{flip}^{\rho'}$

Proof. By inversion:

- $\Pr\left[\hat{b}' \doteq \mathtt{I} \;\middle|\; \Phi\right] = 1/2$
- $\left[\hat{b}' \perp\!\!\!\perp \hat{b}, \Psi^F \setminus \{\hat{b}'\}, (\Psi^B \cup \{\rho_1 \mapsto \{\hat{b}_1\}, \rho_2 \mapsto \{\hat{b}_2\}, \rho_3 \mapsto \{\hat{b}_3\}\})(\{\rho'' \mid \rho'' \sqsubset \rho'\}) \;\middle|\; \Phi\right]$ *(H)*

STS:

- $\left[\hat{b}' \perp\!\!\!\perp \mathrm{bit}(N), \Psi^F \setminus \{\hat{b}'\}, (\Psi^B \cup \{\rho_1 \sqcup \rho_2 \sqcup \rho_3 \mapsto \{\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)\}\})(\{\rho'' \mid \rho'' \sqsubset \rho'\}) \;\middle|\; \Phi\right]$

(1) Case $\rho_1 \sqcup \rho_2 \sqcup \rho_3 \not\sqsubset \rho'$:

$(\Psi^B \cup \{\rho_1 \sqcup \rho_2 \sqcup \rho_3 \mapsto \{\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)\}\})(\{\rho'' \mid \rho'' \sqsubset \rho'\}) = \Psi^B$

By *(H)* and Decomposition

(2) Case $\rho_1 \sqcup \rho_2 \sqcup \rho_3 \sqsubset \rho'$:

$(\Psi^B \cup \{\rho_1 \sqcup \rho_2 \sqcup \rho_3 \mapsto \{\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)\}\})(\{\rho'' \mid \rho'' \sqsubset \rho'\}) = \Psi^B(\{\rho'' \mid \rho'' \sqsubset \rho'\}) \cup \{\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)\}$

$(\Psi^B \cup \{\rho_1 \mapsto \{\hat{b}_1\}, \rho_2 \mapsto \{\hat{b}_2\}, \rho_3 \mapsto \{\hat{b}_3\}\})(\{\rho'' \mid \rho'' \sqsubset \rho'\}) = \Psi^B(\{\rho'' \mid \rho'' \sqsubset \rho'\}) \cup \{\hat{b}_1, \hat{b}_2, \hat{b}_3\}$

By *(H)* and Cond Independence

□

Lemma B.1.1.29 (Type Preservation: Flip)**.**

If: $\Psi^F \uplus \{\hat{b}_3\}, \Psi^B \uplus \{\rho_1 \mapsto \{\hat{b}_1\}\}, \Phi \vdash \hat{b}_2 : \texttt{flip}^{\rho_2}$

And: $\Psi^F \uplus \{\hat{b}_2\}, \Psi^B \uplus \{\rho_1 \mapsto \{\hat{b}_1\}\}, \Phi \vdash \hat{b}_3 : \texttt{flip}^{\rho_3}$

And: $\rho_1 \sqsubset \rho_2$ *and* $\rho_1 \sqsubset \rho_3$

Then:

(1) $\Psi^F \uplus \{\widehat{\text{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)\}, \Psi^B, \Phi \vdash \widehat{\text{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3) : \texttt{flip}^{\rho_2 \sqcap \rho_3}$

(2) $\Psi^F \uplus \{\widehat{\text{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)\}, \Psi^B, \Phi \vdash \widehat{\text{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2) : \texttt{flip}^{\rho_2 \sqcap \rho_3}$

(3) $\forall \rho', \hat{b}'.$

$\Psi^F \setminus \{\hat{b}'\} \uplus \{\hat{b}_2, \hat{b}_3\}, \Psi^B \uplus \{\rho_1 \mapsto \{\hat{b}_1\}\}, \Phi \vdash \hat{b}' : \texttt{flip}^{\rho'}$

$\implies$

$\Psi^F \setminus \{\hat{b}'\} \uplus \{\widehat{\text{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3), \widehat{\text{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)\}, \Psi^B, \Phi \vdash \hat{b}' : \texttt{flip}^{\rho'}$

Proof. By inversion:

- $\Pr\left[\hat{b}_2 \doteq \texttt{I} \;\middle|\; \Phi\right] = 1/2$ *(H11)*
- $\left[\hat{b}_2 \perp\!\!\!\perp \Psi^F \uplus \{\hat{b}_3\}, (\Psi^B \uplus \{\rho_1 \mapsto \{\hat{b}_1\}\})(\{\rho' \mid \rho' \sqsubset \rho_2\}) \;\middle|\; \Phi\right]$ *(H12)*
- $\Pr\left[\hat{b}_3 \doteq \texttt{I} \;\middle|\; \Phi\right] = 1/2$ *(H21)*
- $\left[\hat{b}_3 \perp\!\!\!\perp \Psi^F \uplus \{\hat{b}_2\}, (\Psi^B \uplus \{\rho_1 \mapsto \{\hat{b}_1\}\})(\{\rho' \mid \rho' \sqsubset \rho_3\}) \;\middle|\; \Phi\right]$ *(H22)*

By $\rho_1 \sqsubset \rho_2$ and $\rho_1 \sqsubset \rho_3$:

- $\left[\hat{b}_2 \perp\!\!\!\perp \Psi^F \uplus \{\hat{b}_3\}, \Psi^B(\{\rho' \mid \rho' \sqsubset \rho_2\}), \hat{b}_1 \;\middle|\; \Phi\right]$ *(H13)*
- $\left[\hat{b}_3 \perp\!\!\!\perp \Psi^F \uplus \{\hat{b}_2\}, \Psi^B(\{\rho' \mid \rho' \sqsubset \rho_3\}), \hat{b}_1 \;\middle|\; \Phi\right]$ *(H23)*

(1) STS:

(a) $\Pr\left[\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3) \doteq \mathtt{I}\right] = 1/2$ *(i)*

(b) $\left[\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3) \perp\!\!\!\perp \Psi^F \uplus \{\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)\}, \Psi^B(\{\rho' \mid \rho' \sqsubset \rho_2 \sqcap \rho_3\}) \;\middle|\; \Phi\right]$ *(ii)*

(i) is by Cond Stability applied to *(H13)* and Decomposition (to achieve $\left[\hat{b}_2 \perp\!\!\!\perp \hat{b}_1 \;\middle|\; \Phi\right]$), *(H23)* and Decomposition (to achieve $\left[\hat{b}_3 \perp\!\!\!\perp \hat{b}_1 \;\middle|\; \Phi\right]$), *(H11)* and *(H21)*

(ii) is by:

$$
\begin{aligned}
& \Pr\left[\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3) \;\middle|\; \Psi^F \uplus \{\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)\}, \Psi^B(\{\rho' \mid \rho' \sqsubset \rho_2 \sqcap \rho_3\}), \Phi\right] \\
= \;& \wr \text{ Total Probability } \int \\
& \Pr\left[\hat{b}_2 \;\middle|\; \hat{b}_1 \doteq \mathtt{I}, \Psi^F \uplus \{\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)\}, \Psi^B(\{\rho' \mid \rho' \sqsubset \rho_2 \sqcap \rho_3\}), \Phi\right] \Pr\left[\hat{b}_1 \doteq \mathtt{I}\right] \\
& + \\
& \Pr\left[\hat{b}_3 \;\middle|\; \hat{b}_1 \doteq \mathtt{O}, \Psi^F \uplus \{\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)\}, \Psi^B(\{\rho' \mid \rho' \sqsubset \rho_2 \sqcap \rho_3\}), \Phi\right] \Pr\left[\hat{b}_1 \doteq \mathtt{O}\right] \\
= \;& \wr \textit{(H13)}, \textit{(H23)} \text{ and Decomposition } \int \\
& \Pr\left[\hat{b}_2 \;\middle|\; \hat{b}_1 \doteq \mathtt{I}, \Phi\right] \Pr\left[\hat{b}_1 \doteq \mathtt{I}\right] + \Pr\left[\hat{b}_3 \;\middle|\; \hat{b}_1 \doteq \mathtt{O}, \Phi\right] \Pr\left[\hat{b}_1 \doteq \mathtt{O}\right] \\
= \;& \wr \text{ Total Probability } \int \\
& \Pr\left[\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3) \;\middle|\; \Phi\right]
\end{aligned}
$$

(2) STS:

(a) $\Pr\left[\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2) \doteq \mathtt{I}\right] = 1/2$

(b) $\left[\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2) \perp\!\!\!\perp \Psi^F \uplus \{\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)\}, \Psi^B(\{\rho' \mid \rho' \sqsubset \rho_2 \sqcap \rho_3\}) \;\middle|\; \Phi\right]$

Analogous to previous cases

(3) Assume ρ' and $\hat{b}'$ where $\Psi^F \setminus \{\hat{b}'\} \uplus \{\hat{b}_2, \hat{b}_3\}, \Psi^B \uplus \{\rho_1 \mapsto \{\hat{b}_1\}\}, \Phi \vdash \hat{b}' : \mathtt{flip}^{\rho'}$

By inversion:

- $\Pr\left[\hat{b}' \doteq \mathtt{I}\right] = 1/2$
- $\left[\hat{b}' \perp\!\!\!\perp \Psi^F \setminus \{\hat{b}'\} \uplus \{\hat{b}_2, \hat{b}_3\}, (\Psi^B \cup \{\rho_1 \mapsto \{\hat{b}_1\}\})(\{\rho'' \mid \rho'' \sqsubset \rho'\}) \;\middle|\; \Phi\right]$ *(i)*

STS:

- $\left[\hat{b}' \perp\!\!\!\perp \Psi^F \setminus \{\hat{b}'\} \uplus \{\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3), \widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)\}, \Psi^B(\{\rho'' \mid \rho'' \sqsubset \rho'\}) \;\middle|\; \Phi\right]$

$$
\begin{aligned}
& \Pr\left[\hat{b}' \;\middle|\; \Psi^F \setminus \{\hat{b}'\} \uplus \{\widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3), \widehat{\mathrm{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2)\}, \Psi^B(\{\rho'' \mid \rho'' \sqsubset \rho'\}), \Phi\right] \\
= \;& \wr \text{ Total Probability } \int \\
& \Pr\left[\hat{b}' \;\middle|\; \hat{b}_1 \doteq \mathtt{I}, \Psi^F \setminus \{\hat{b}'\} \uplus \{\hat{b}_2, \hat{b}_3\}, \Psi^B(\{\rho'' \mid \rho'' \sqsubset \rho'\}), \Phi\right] \Pr\left[\hat{b}_1 \doteq \mathtt{I}\right] \\
& + \\
& \Pr\left[\hat{b}' \;\middle|\; \hat{b}_1 \doteq \mathtt{O}, \Psi^F \setminus \{\hat{b}'\} \uplus \{\hat{b}_2, \hat{b}_3\}, \Psi^B(\{\rho'' \mid \rho'' \sqsubset \rho'\}), \Phi\right] \Pr\left[\hat{b}_1 \doteq \mathtt{O}\right] \\
= \;& \wr \textit{ (i), (H13) and (H23) } \int \\
& \Pr\left[\hat{b}' \;\middle|\; \hat{b}_1 \doteq \mathtt{I}, \Phi\right] \Pr\left[\hat{b}_1 \doteq \mathtt{I}\right] + \Pr\left[\hat{b}' \;\middle|\; \hat{b}_1 \doteq \mathtt{O}, \Phi\right] \Pr\left[\hat{b}_1 \doteq \mathtt{O}\right] \\
= \;& \wr \text{ Total Probability } \int \\
& \Pr\left[\hat{b}' \;\middle|\; \Phi\right]
\end{aligned}
$$

□

Lemma B.1.1.30 (Type Preservation: Substitution)**.**

If: $\mathcal{K}(\tau_1) = \mathtt{A}$

And: $\Psi_c \uplus \Psi_2, \Phi, \Sigma, \varnothing \vdash \underline{v} : \tau_1, \Psi_1$

And: $\Psi_c \uplus \Psi_1, \Phi, \Sigma, \Gamma \uplus [x \mapsto \tau_1] \vdash \underline{e} : \tau_2 \,;\, \Gamma' \uplus [x \mapsto \overset{\bullet}{\tau}{}'_1], \Psi_2$

Then: *there exists* Ψ'_1

S.t.: $\Psi'_1 \subseteq \Psi_1$

And: $\Psi_c, \Phi, \Sigma, \Gamma \vdash [\underline{v}/x]\underline{e} : \tau_2 \,;\, \Gamma', \Psi'_1 \uplus \Psi_2$

Proof.

Case analysis on $\overset{\bullet}{\tau}{}'_1$:

(1) $\overset{\bullet}{\tau}{}'_1 = \tau_1$

$\Psi'_1 \triangleq \varnothing$

By $\varnothing \subseteq \Psi_1$, Affine Substitution Unused and Weaken Expression applied to Weaken Bit Value and Weaken Flip

(2) $\overset{\bullet}{\tau}{}'_1 = \bullet$

$\Psi_1' \triangleq \Psi_1$

By $\Psi_1 \subseteq \Psi_1$ and Affine Substitution Used

□

Lemma B.1.1.31 (Affine Substitution Used)**.**

If: $\mathcal{K}(\tau_1) = \mathtt{A}$

And: $\Psi_c \uplus \Psi_2, \Phi, \Sigma, \varnothing \vdash \underline{v} : \tau_1 \; ; \; \varnothing, \Psi_1$

And: $\Psi_c \uplus \Psi_1, \Phi, \Sigma, \Gamma \uplus [x \mapsto \tau_1] \vdash \underline{e} : \tau_2 \; ; \; \Gamma' \uplus [x \mapsto \bullet], \Psi_2$

Then: $\Psi_c, \Phi, \Sigma, \Gamma \vdash [\underline{v}/x]\underline{e} : \tau_2 \; ; \; \Gamma', \Psi_1 \uplus \Psi_2$

Proof.

Induction on $\underline{e}$, Context Monotonicity and Affine Substitution Unused

Representative inductive case:

$\underline{e} = \langle \underline{e}_1, \underline{e}_2 \rangle$

Must be one of the following (by Context Monotonicity):

(1)
$$\frac{\Psi_c \uplus \Psi_1 \uplus \Psi_{22}, \Phi, \Sigma, \Gamma \uplus [x \mapsto \tau_1] \vdash \underline{e}_{21} : \tau_{21} \; ; \; \Gamma'' \uplus [x \mapsto \tau_1], \Psi_{21} \qquad \Psi_c \uplus \Psi_1 \uplus \Psi_{21}, \Phi, \Sigma, \Gamma'' \uplus [x \mapsto \tau_1] \vdash \underline{e}_{22} : \tau_{22} \; ; \; \Gamma' \uplus [x \mapsto \bullet], \Psi_{22}}{\Psi_c \uplus \Psi_1, \Phi, \Sigma, \Gamma \uplus [x \mapsto \tau_1] \vdash \langle \underline{e}_{21}, \underline{e}_{22} \rangle : \tau_{21} \times \tau_{22} \; ; \; \Gamma' \uplus [x \mapsto \bullet], \Psi_{21} \uplus \Psi_{22}}$$

Goal: $\Psi_c, \Phi, \Sigma, \Gamma \uplus [x \mapsto \tau_1] \vdash [\underline{v}/x]\langle \underline{e}_{21}, \underline{e}_{22} \rangle : \tau_{21} \times \tau_{22} \; ; \; \Gamma' \uplus [x \mapsto \bullet], \Psi_1 \uplus \Psi_{21} \uplus \Psi_{22}$

$[\underline{v}/x]\langle \underline{e}_{21}, \underline{e}_{22} \rangle = \langle \underline{e}_{21}, [\underline{v}/x]\underline{e}_{22} \rangle$ (by Affine Substitution Unused)

STS:

(a) $\Psi_c \uplus \Psi_{22}, \Phi, \Sigma, \Gamma \vdash \underline{e}_{21} : \tau_{21} \; ; \; \Gamma'', \Psi_{21}$ (by Weaken Expression)

(b) $\Psi_c \uplus \Psi_{21}, \Phi, \Sigma, \Gamma'' \vdash [\underline{v}/x]\underline{e}_{22} : \tau \; ; \; \Gamma', \Psi_1 \uplus \Psi_{22}$ (by Inductive Hypothesis)

(2)
$$\frac{\Psi_c \uplus \Psi_1 \uplus \Psi_{22}, \Phi, \Sigma, \Gamma \uplus [x \mapsto \tau_1] \vdash \underline{e}_{21} : \tau_{21} \; ; \; \Gamma'' \uplus [x \mapsto \bullet], \Psi_{21} \qquad \Psi_c \uplus \Psi_1 \uplus \Psi_{21}, \Phi, \Sigma, \Gamma'' \uplus [x \mapsto \bullet] \vdash \underline{e}_{22} : \tau_{22} \; ; \; \Gamma' \uplus [x \mapsto \bullet], \Psi_{22}}{\Psi_c, \Phi, \Sigma, \Gamma \uplus [x \mapsto \tau_1] \vdash \langle \underline{e}_{21}, \underline{e}_{22} \rangle : \tau_{21} \times \tau_{22} \; ; \; \Gamma' \uplus [x \mapsto \bullet], \Psi_{21} \uplus \Psi_{22}}$$

Analogous to (1) where $[\underline{v}/x]\langle \underline{e}_{21}, \underline{e}_{22}\rangle = \langle [\underline{v}/x]\underline{e}_{21}, \underline{e}_{22}\rangle$

□

Lemma B.1.1.32 (Affine Substitution Unused)**.**

If: $\Psi_c, \Phi, \Sigma, \Gamma \uplus [x \mapsto \bullet] \vdash \underline{e} : \tau_2 \; ; \; \Gamma' \uplus [x \mapsto \bullet], \Psi$

Or: $\mathcal{K}(\tau_1) = \mathtt{A}$ *and* $\Psi_c, \Phi, \Sigma, \Gamma \uplus [x \mapsto \tau_1] \vdash \underline{e} : \tau_2 \; ; \; \Gamma' \uplus [x \mapsto \tau_1], \Psi$

Then: $[\underline{v}/x]\underline{e} = \underline{e}$

Proof.

Induction on $\underline{e}$ and Context Monotonicity

□

Lemma B.1.1.33 (Context Monotonicity)**.**

If: $\Psi_c, \Phi, \Sigma, \Gamma \vdash \underline{e} : \tau_2 \; ; \; \Gamma', \Psi$

Then: $\Gamma(x) \sqsubseteq \Gamma'(x)$

Proof.

Induction on $\underline{e}$ and partial order properties

□

Weakening

Lemma B.1.1.34 (Weaken Context)**.**

If: $\Psi_c, \Phi, \Sigma \vdash \underline{\sigma}, \underline{e} : \tau, \Psi$

And: $\Psi_c, \Phi', \Sigma' \vdash \underline{\sigma'}, \underline{e'} : \tau, \Psi'$

And: $\Sigma' \supseteq \Sigma$

And: $\forall \hat{b}, \rho.\ \Psi \uplus \Psi_c, \Phi \vdash \hat{b} : \mathtt{bit}^{\rho}_{S} \implies \Psi' \uplus \Psi_c, \Phi' \vdash \hat{b} : \mathtt{bit}^{\rho}_{S}$

And: $\forall \hat{b}, \rho.\ \Psi/\Psi_c, \Phi \vdash \hat{b} : \mathtt{flip}^{\rho} \implies \Psi'/\Psi_c, \Phi' \vdash \hat{b} : \mathtt{flip}^{\rho}$

And: $\Phi, \Sigma \vdash \underline{\sigma}, \underline{E}[\underline{e}] : \tau', \Psi_c \uplus \Psi$

Then: $\Phi', \Sigma' \vdash \underline{\sigma'}, \underline{E}[\underline{e'}] : \tau', \Psi_c \uplus \Psi'$

Proof.

Induction on $\underline{E}$ and Weaken Expression

□

Lemma B.1.1.35 (Weaken Store)**.**

If: $\Psi_c, \Phi, \Sigma \vdash \underline{\sigma}\ ;\ \Psi$

And: $\Sigma' \supseteq \Sigma$

And: $\forall \hat{b}, \rho.\ \Psi_c/\Psi, \Phi \vdash \hat{b} : \mathtt{flip}^{\rho} \implies \Psi'_c/\Psi, \Phi' \vdash \hat{b} : \mathtt{flip}^{\rho}$

Then: $\Psi'_c, \Phi', \Sigma' \vdash \underline{\sigma}\ ;\ \Psi$

Proof.

Induction on $\underline{\sigma}$, Weaken Expression and $\Sigma(\iota) = \tau \implies \Sigma'(\iota) = \tau$

□

Lemma B.1.1.36 (Weaken Expression)**.**

If: $\Psi_c, \Phi, \Sigma, \Gamma \vdash \underline{e} : \tau \; ; \; \Gamma', \Psi$

And: $\Sigma' \supseteq \Sigma$

And: $\forall \hat{b}, \rho.\ \Psi_c/\Psi, \Phi \vdash \hat{b} : \mathtt{flip}^{\rho} \implies \Psi'_c/\Psi, \Phi' \vdash \hat{b} : \mathtt{flip}^{\rho}$ *(H)*

Then: $\Psi'_c, \Phi', \Sigma', \Gamma \vdash \underline{e} : \tau \; ; \; \Gamma', \Psi$

Induction on $\underline{e}$, Weaken Bit Value and application of *(H)* on flip values
Lemma B.1.1.37 (Weaken Bit Value)**.**

If: $\Psi_c, \Phi \vdash \hat{b} : \mathtt{bitv}^{\rho}_{\ell}$

Then: $\Psi'_c, \Phi' \vdash \hat{b} : \mathtt{bit}^{\rho}_{\ell}$

Proof.

Immediate by inversion and re-construction of the type derivation

□

Lemma B.1.1.38 (Weaken Flip)**.**

If: $\Psi^F_c, \Psi^B_c, \Phi \vdash \hat{b} : \mathtt{flipv}^{\rho}$

And: $\Psi^{F\prime}_c, \Psi^{B\prime}_c \subseteq \Psi^F_c, \Psi^B_c$

Then: $\Psi^{F\prime}_c, \Psi^B_c, \Phi \vdash \hat{b} : \mathtt{flipv}^{\rho}$

Proof.

By inversion:

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

STS: $\left[\hat{b} \perp\!\!\!\perp \Psi^{F\prime}_c, \Psi^{B\prime}_c(\{\rho' \mid \rho' \sqsubset \rho\}) \;\middle|\; \Phi\right]$

By *(H)* and Decomposition with $\Psi^{F\prime}_c \subseteq \Psi^F_c$ and $\Psi^{B\prime}_c(\{\rho' \mid \rho' \sqsubset \rho\}) \subseteq \Psi^B_c(\{\rho' \mid \rho' \sqsubset \rho\})$

□

Intensional Distribution Lemmas

All of the following lemmas are proved for intensional distributions $\hat{x} \in \mathcal{I}(A)$, however except for Monad Idempotence (Intensional Only), each of the properties are also true of denotational distributions $\tilde{x} \in \mathcal{D}(A)$ (although the proof given only applies to intensional distributions). Recall that trees are considered equal $=$ when they are syntactically equal modulo height extension, i.e., $\hat{x} = \langle \hat{x} \hat{x} \rangle$.

Lemma B.1.1.39 (Proper Distribution)**.**

(1) $\sum_{x \in \text{support}(\hat{x})} \Pr[\hat{x} \doteq x] = 1$

(2) If: $\Pr[\hat{y} \doteq y] > 0$

Then: $\Pr[\hat{x} \doteq x \mid \hat{y} \doteq y]$ *is defined*

And: $\sum_{x \in \text{support}(\hat{x})} \Pr[\hat{x} \doteq x \mid \hat{y} \doteq y] = 1$

Proof. Induction on the tree-structure of $\hat{x}$ □

Lemma B.1.1.40 (Return Probability)**.**

(1) $\Pr[\text{return}_{\mathcal{I}}(x) \doteq x] = 1$

(2) $\Pr[\text{return}_{\mathcal{I}}(x) \doteq y] = 0$ *when* $x \neq y$

Proof. Immediate by definition of return and Pr

□

Lemma B.1.1.41 (Bind Probability)**.**

$$\Pr[\text{do } x \leftarrow \hat{x} \,;\, f(x) \doteq y] = \sum_{x} \Pr[f(x) \doteq y \mid \hat{x} \doteq x] \Pr[\hat{x} \doteq x]$$

Proof. Induction on the tree-structure of $\hat{x}$

□

Lemma B.1.1.42 (Monad Laws)**.**

$(\text{do } x \leftarrow \text{return}_{\mathcal{I}}(y) \; ; \; f(x)) = f(y)$ *(left-unit)*

$(\text{do } x \leftarrow \hat{x} \; ; \; \text{return}(x)) = \hat{x}$ *(right-unit)*

$(\text{do } y \leftarrow (\text{do } x \leftarrow \hat{x} \; ; \; f(x)) \; ; \; g(y)) = (\text{do } x \leftarrow \hat{x} \; ; \; y \leftarrow f(x) \; ; \; g(y))$ *(associativity)*

Proof.(1) (left-unit)

immediate from definitions

(2) (right-unit)

Case analysis on $\hat{x}$

- Case $\hat{x} = x$:

  $x = x$; immediate

- Case $\hat{x} = \langle \hat{x}_1 \hat{x}_2 \rangle$:

  $\langle \pi_1(\langle \hat{x}_1 \hat{x}_2 \rangle) \pi_2(\langle \hat{x}_1 \hat{x}_2 \rangle) \rangle = \langle \hat{x}_1 \hat{x}_2 \rangle$; immediate

(3) (associativity)

Case analysis on $\hat{x}$:

- Case $\hat{x} = x$:

  $(\text{do } y \leftarrow f(x) \; ; \; g(y)) = (\text{do } y \leftarrow f(x) \; ; \; g(y))$; immediate

- Case $\hat{x} = \langle \hat{x}_1 \hat{x}_2 \rangle$:

  $\langle \pi_1(g(\pi_1(f(\hat{x}_1)))) \pi_2(g(\pi_2(f(\hat{x}_2)))) \rangle = \langle \pi_1(g(\pi_1(f(\hat{x}_1)))) \pi_2(g(\pi_2(f(\hat{x}_2)))) \rangle$; immediate

□

Lemma B.1.1.43 (Monad Commutativity)**.**

$(\text{do } x \leftarrow \hat{x} \; ; \; y \leftarrow \hat{y} \; ; \; f(x, y)) = (\text{do } y \leftarrow \hat{y} \; ; \; x \leftarrow \hat{x} \; ; \; f(x, y))$

Proof. Case analysis on $\hat{x}$:

- Case $\hat{x} = x$:

$(\text{do } y \leftarrow \hat{y} \; ; \; f(x, y)) = (\text{do } y \leftarrow \hat{y} \; ; \; f(x, y))$; immediate

- Case $\hat{x} = \langle \hat{x}_1 \hat{x}_2 \rangle$:

  $\langle \pi_1(\text{do } y \leftarrow \hat{y} \; ; \; f(\hat{x}_1, y)) \pi_2(\text{do } y \leftarrow \hat{y} \; ; \; f(\hat{x}_2, y)) \rangle$

  $=$

  $\text{do } y \leftarrow \hat{y} \; ; \; \langle \pi_1(f(\hat{x}_1, y)) \pi_2(f(\hat{x}_2, y)) \rangle$

  Finally by case analysis on $\hat{y}$

□

Lemma B.1.1.44 (Monad Idempotence (Intensional Only))**.**

The intensional distribution monad $\mathcal{I}$ is idempotent.

NOTE: this is in contrast with the denotational distribution monad $\mathcal{D}$ which is not idempotent.

$(\text{do } x_1 \leftarrow \hat{x} \; ; \; x_2 \leftarrow \hat{x} \; ; \; f(x_1, x_2)) = (\text{do } x \leftarrow \hat{x} \; ; \; f(x, x))$

Proof. Case analysis on $\hat{x}$ (analogous to monad laws and commutativity proofs)

□

Lemma B.1.1.45 (Bit Independence)**.**

A particular random bit is independent of all other random bits.

$\text{bit}(N) \perp\!\!\!\perp \text{bit}(N')$ *for* $N \neq N'$

Proof. Induction on N and N'

□

Lemma B.1.1.46 (Cond Independence)**.**

A conditional is independent when its inputs are jointly indpendent.

$\hat{x} \perp\!\!\!\perp \hat{b}, \hat{y}, \hat{z} \implies \hat{x} \perp\!\!\!\perp \text{cond}(\hat{b}, \hat{y}, \hat{z})$

Proof. By Total Probability on $\hat{b}$ and unfolding definition of $\hat{b}$

□

Lemma B.1.1.47 (Cond Stability)**.**

A conditional is stable when the guard is independent of branches, and branches have equal distributions.

If: $\hat{b} \perp\!\!\!\perp \hat{x}_1$

And: $\hat{b} \perp\!\!\!\perp \hat{x}_2$

And: $\Pr[\hat{x}_1 \doteq x] = \Pr[\hat{x}_2 \doteq x]$

Then:

(1) $\Pr\left[\text{cond}(\hat{b}, \hat{x}_1, \hat{x}_2) \doteq x\right] = \Pr[\hat{x}_1 \doteq x] = \Pr[\hat{x}_2 \doteq x]$

(2) $\hat{b} \perp\!\!\!\perp \text{cond}(\hat{b}, \hat{x}_1, \hat{x}_2)$

Proof.(1)

$$
\begin{aligned}
& \Pr\left[\text{cond}(\hat{b}, \hat{x}_1, \hat{x}_2) \doteq x \;\middle|\; \hat{b} \doteq b\right] \\
= \;& \wr \;\; \text{Total Probability} \;\; \int \\
& \Pr\left[\hat{x}_1 \doteq x \;\middle|\; \hat{b} \doteq \mathtt{I}\right] \Pr\left[\hat{b} \doteq \mathtt{I}\right] + \Pr\left[\hat{x}_2 \doteq x \;\middle|\; \hat{b} \doteq \mathtt{O}\right] \Pr\left[\hat{b} \doteq \mathtt{O}\right] \\
= \;& \wr \;\; \hat{b} \perp\!\!\!\perp \hat{x}_i \;\; \int \\
& \Pr[\hat{x}_1 \doteq x] \Pr\left[\hat{b} \doteq \mathtt{I}\right] + \Pr[\hat{x}_2 \doteq x] \Pr\left[\hat{b} \doteq \mathtt{O}\right] \\
= \;& \wr \;\; \Pr[\hat{x}_1 \doteq x] = \Pr[\hat{x}_2 \doteq x] \;\; \int \\
& \Pr[\hat{x}_1 \doteq x] \left(\Pr\left[\hat{b} \doteq \mathtt{I}\right] + \Pr\left[\hat{b} \doteq \mathtt{O}\right]\right) \\
= \;& \wr \;\; \text{Proper Distribution} \;\; \int \\
& \Pr[\hat{x}_1 \doteq x] \\
= \;& \Pr[\hat{x}_2 \doteq x]
\end{aligned}
$$

(2) Follows direction from (1)

□

Probability Facts

All of the following facts are stated using intensional distribution notation $\hat{x} \in \mathcal{I}(A)$, however they are true of any model which supports joint probabilities, including $\tilde{x} \in \mathcal{D}(A)$. Proofs are not given because they are standard properties w.r.t. standard definitions.

Fact B.1.1 (Conditional Decomposition)**.**

$$\Pr\left[\hat{x} \doteq x \mid \hat{y} \doteq y\right] = \frac{\Pr[\hat{x} \doteq x, \hat{y} \doteq y]}{\Pr[\hat{y} \doteq y]}$$

Fact B.1.2 (Bayes' Rule)**.**

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

Fact B.1.3 (Total Probability)**.**

$$\Pr\left[\hat{x} \doteq x\right] = \sum_{y \in \text{support}(\hat{y})} \Pr\left[\hat{x} \doteq x \mid \hat{y} \doteq y\right] \Pr\left[\hat{y} \doteq y\right]$$

Proof. Induction on the tree-structure of $\hat{x}$ and $\hat{y}$ □

Fact B.1.4 (Decomposition)**.**

(1) $\hat{x} \perp\!\!\!\perp \hat{y}, \hat{z} \implies \hat{x} \perp\!\!\!\perp \hat{y}$

(2) $\hat{x} \perp\!\!\!\perp \hat{y}, \hat{z} \implies \hat{x} \perp\!\!\!\perp \hat{z}$

Fact B.1.5 (Decomposition)**.**

(1) $\hat{x} \perp\!\!\!\perp \hat{y}, \hat{z} \implies [\hat{x} \perp\!\!\!\perp \hat{y} \mid \hat{z}]$

(2) $\hat{x} \perp\!\!\!\perp \hat{y}, \hat{z} \implies [\hat{x} \perp\!\!\!\perp \hat{z} \mid \hat{y}]$

Fact B.1.6 (Decomposition)**.**

If: $\hat{x} \perp\!\!\!\perp \hat{y}$

And: $[\hat{x} \perp\!\!\!\perp \hat{y} \mid \hat{z}]$

Then: $\hat{x} \perp\!\!\!\perp \hat{y}, \hat{z}$

Fact B.1.7 (Independence Equivalences)**.**

(1) $\left[\hat{\overline{x}} \perp\!\!\!\perp \hat{\overline{y}} \mid \hat{\overline{z}} \doteq z\right]$

$\overset{\triangle}{\iff} \forall \overline{x}, \overline{y}.\ \Pr\left[\hat{\overline{x}} \doteq \overline{x}, \hat{\overline{y}} \doteq \overline{y}, \hat{\overline{z}} \doteq z\right] = \Pr\left[\hat{\overline{x}} \doteq \overline{x} \mid \hat{\overline{z}} \doteq z\right] \Pr\left[\hat{\overline{y}} \doteq \overline{y} \mid \hat{\overline{z}} \doteq z\right]$

$\iff \forall \overline{x}, \overline{y}.\ \Pr\left[\hat{\overline{x}} \doteq \overline{x} \mid \hat{\overline{y}} \doteq \overline{y}, \hat{\overline{z}} \doteq z\right] = \Pr\left[\hat{\overline{x}} \doteq \overline{x} \mid \hat{\overline{z}} \doteq z\right]$

$\iff \forall \overline{x}, \overline{y}.\ \Pr\left[\hat{\overline{y}} \doteq \overline{y} \mid \hat{\overline{x}} \doteq \overline{x}, \hat{\overline{z}} \doteq z\right] = \Pr\left[\hat{\overline{y}} \doteq \overline{y} \mid \hat{\overline{z}} \doteq z\right]$

$\iff \left[\hat{\overline{y}} \perp\!\!\!\perp \hat{\overline{x}} \mid \hat{\overline{z}} \doteq z\right]$

Fact B.1.8 (Return Equivalence)**.**

If: $x_1 \sim_A x_2$

Then: $\left[\text{return}_{\mathcal{I}}(x_1) \mid \hat{\overline{y}} \doteq y\right] \approx_{\sim_A} \left[\text{return}_{\mathcal{I}}(x_2) \mid \hat{\overline{z}} \doteq z\right]$

Fact B.1.9 (Bind Equivalence)**.**

For: $f_1, f_2 \in A \to \mathcal{I}(B)$

If: $\hat{x}_1 \approx_{\sim_A} \hat{x}_2$

And:

$\forall (x_1 \in \text{support}(\hat{x}_1)), (x_2 \in \text{support}(\hat{x}_2)).\ x_1 \sim_A x_2 \implies [f_1(x_1) \mid \hat{x}_1 \doteq x_1] \approx_{\sim_B} [f_2(x_2) \mid \hat{x}_2 \doteq x_2]$

Then: $(\hat{x}_1 \gg\!= f_1) \approx_{\sim_B} (\hat{x}_2 \gg\!= f_2)$

Fact B.1.10 (Extensional Equivalence)**.**

$$\left[\hat{\overline{x}_1} \mid \hat{\overline{y}} \doteq y\right] \approx_{=} \left[\hat{\overline{x}_2} \mid \hat{\overline{z}} \doteq z\right]$$

$$\iff$$

$$\forall \overline{x}, \overline{y}.\ \Pr\left[\hat{\overline{x}_1} \doteq \overline{x} \mid \hat{\overline{y}} \doteq y\right] = \Pr\left[\hat{\overline{x}_2} \doteq \overline{x} \mid \hat{\overline{z}} \doteq z\right]$$

Fact B.1.11 (Distribution Equality Injective Function)**.**

If: $\hat{x}_1 \approx_{=} \hat{x}_2$

And: f *is injective*

Then: $(\text{do } x \leftarrow \hat{x}_1 \,;\, \text{return}(f(x))) \approx_{=} (\text{do } x \leftarrow \hat{x}_2 \,;\, \text{return}(f(x)))$

B.1.3 Definitions

$\boxed{\mathcal{D}(A) \in \text{set}}$

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

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

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

$$\left[\tilde{x}_1 \;\middle|\; \widetilde{y \doteq y}\right] \approx_{\sim_A} \left[\tilde{x}_2 \;\middle|\; \widetilde{z \doteq z}\right] \overset{\triangle}{\iff} \forall x. \left(\sum_{x' | x' \sim_A x} \Pr\left[\tilde{x}_1 \doteq x' \;\middle|\; \widetilde{y \doteq y}\right] \right) = \left(\sum_{x' | x' \sim_A x} \Pr\left[\tilde{x}_2 \doteq x' \;\middle|\; \widetilde{z \doteq z}\right] \right)$$

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

$\boxed{\mathcal{I}(A) \in \text{set}}$

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

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

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

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

$$\begin{array}{rcl} \pi_2 & \in & \mathcal{I}(A) \to \mathcal{I}(A) \end{array}$$

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

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

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

$$\begin{array}{rcl} \text{return} & \in & A \to \mathcal{I}(A) \\ \text{return}(x) & \triangleq & x \end{array}$$

$$\begin{array}{rcl} \text{bind} & \in & \mathcal{I}(A) \times (A \to \mathcal{I}(B)) \to \mathcal{I}(B) \\ \text{bind}(x, f) & \triangleq & f(x) \end{array}$$

								e	$\in$	exp	
ℓ	$\in$	label	$::=$	P | S	public and secret			e	$::=$	v	value expressions
				(*where* P $\sqsubset$ S)	security labels				|	b_ℓ	bit literal
ρ	$\in$	R			probability region				|	$\texttt{flip}^\rho()$	coin flip in region
b	$\in$	$\mathbb{B}$	$::=$	$\texttt{0} \mid \texttt{1}$	bits				|	$\texttt{cast}_\ell(v)$	cast flip to bit
x, y	$\in$	var			variables				|	$\texttt{mux}(e,e,e)$	atomic conditional
v	$\in$	val	$::=$	x	variable values				|	$\texttt{xor}(e,e)$	bit xor
			|	$\langle v, v\rangle$	tuple values				|	$\texttt{if}(e)\{e\}\{e\}$	branch conditional
			|	$\texttt{fun}_y(x{:}\tau).e$	function values				|	$\texttt{ref}(e)$	reference creation
τ	$\in$	type	$::=$	$\texttt{bit}_\ell^\rho$	non-random bit				|	$\texttt{read}(e)$	reference read
			|	$\texttt{flip}^\rho$	secret uniform bit				|	$\texttt{write}(e,e)$	reference write
			|	$\texttt{ref}(\tau)$	reference				|	$\langle e, e\rangle$	tuple creation
			|	$\tau \times \tau$	tuple				|	$\texttt{let}\ x = e\ \texttt{in}\ e$	variable binding
			|	$\tau \to \tau$	function				|	$\texttt{let}\ x, y = e\ \texttt{in}\ e$	tuple elimination
									|	$e(e)$	fun. application

ι	$\in$	loc	$\approx$	$\mathbb{N}$				$\underline{e}$	$\in$	$\underline{\text{exp}}$	$::=$	$\ldots$	same schema
v	$\in$	val	$::=$	$\ldots$				$\underline{E}$	$\in$	$\underline{\text{cxt}}$	$::=$	$\ldots$	same schema
			|	$\texttt{bitv}_\ell(b)$				$\underline{v}$	$\in$	$\underline{\text{val}}$	$::=$	$\ldots$	extended with…
			|	$\texttt{flipv}(b)$							|	$\texttt{bitv}_\ell(\hat{b})$	bit value
			|	$\texttt{locv}(\iota)$							|	$\texttt{flipv}(\hat{b})$	flip value
σ	$\in$	store	$\triangleq$	loc $\rightharpoonup$ val							|	$\texttt{locv}(\iota)$	location value
ς	$\in$	config	$::=$	σ, e				$\underline{\sigma}$	$\in$	$\underline{\text{store}}$	$\triangleq$	loc $\rightharpoonup$ $\underline{\text{val}}$	store
t	$\in$	trace	$::=$	$\epsilon \mid t{\cdot}\varsigma$				$\underline{\varsigma}$	$\in$	$\underline{\text{config}}$	$::=$	$\underline{\sigma}, \underline{e}$	configuration
								$\underline{t}$	$\in$	$\underline{\text{trace}}$	$::=$	$\epsilon \mid \underline{t}{\cdot}\underline{\varsigma}$	trace

E	$\in$	cxt	$::=$	$\square$			$\overset{\bullet}{e}$	$\in$	$\overset{\bullet}{\text{exp}}$	$::=$	$\ldots$	same schema
			|	$\texttt{mux}(E,e,e) \mid \texttt{mux}(v,E,e) \mid \texttt{mux}(v,v,E)$			$\overset{\bullet}{E}$	$\in$	$\overset{\bullet}{\text{cxt}}$	$::=$	$\ldots$	same schema
			|	$\texttt{xor}(E,e) \mid \texttt{xor}(v,E) \mid \texttt{if}(E,e,e)$			$\overset{\bullet}{v}$	$\in$	$\overset{\bullet}{\text{val}}$	$::=$	$\ldots \mid \bullet$	extend with $\bullet$
			|	$\texttt{ref}(E) \mid \texttt{read}(E) \mid \texttt{write}(E,e) \mid \texttt{write}(v,E)$			$\overset{\bullet}{\sigma}$	$\in$	$\overset{\bullet}{\text{store}}$	$\triangleq$	loc $\rightharpoonup$ $\overset{\bullet}{\text{val}}$	store
			|	$\langle E,e\rangle \mid \langle v,E\rangle \mid E(e) \mid v(E)$			$\overset{\bullet}{\varsigma}$	$\in$	$\overset{\bullet}{\text{config}}$	$::=$	$\overset{\bullet}{\sigma}, \overset{\bullet}{e}$	configuration
			|	$\texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{let}\ x, y = E\ \texttt{in}\ e$			$\overset{\bullet}{t}$	$\in$	$\overset{\bullet}{\text{trace}}$	$::=$	$\epsilon \mid \overset{\bullet}{t}, \overset{\bullet}{\varsigma}$	trace

Figure B.2: (1) Source Syntax; and (2) Runtime syntax: standard, mixed and adver-

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

$$\kappa \in \text{kind} ::= \mathtt{U} \mid \mathtt{A} \quad (\textit{where } \mathtt{U} \sqsubset \mathtt{A}) \qquad (\Gamma_1 \sqcup \Gamma_2)(x) \triangleq \Gamma_1(x) \sqcup \Gamma_2(x)$$

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

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

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

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

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

Bit
$$\frac{}{\Gamma \vdash b_\ell : \mathtt{bit}_\ell^{\varnothing} \dashv \Gamma}$$

Flip
$$\frac{\rho \neq \bot}{\Gamma \vdash \mathtt{flip}^\rho() : \mathtt{flip}^\rho \dashv \Gamma}$$

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

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

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

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

Mux-Flip
$$\frac{\Gamma \vdash e_1 : \mathtt{bit}_{\ell_1}^{\rho_1} \dashv \Gamma' \quad \rho_1 \sqsubset \rho_2 \quad \Gamma' \vdash e_2 : \mathtt{flip}^{\rho_2} \dashv \Gamma'' \quad \rho_1 \sqsubset \rho_3 \quad \Gamma'' \vdash e_3 : \mathtt{flip}^{\rho_3} \dashv \Gamma''' \quad \rho = \rho_2 \sqcap \rho_3}{\Gamma \vdash \mathtt{mux}(e_1, e_2, e_3) : \mathtt{flip}^\rho \times \mathtt{flip}^\rho \dashv \Gamma'''}$$

Xor-Flip
$$\frac{\Gamma \vdash e_1 : \mathtt{bit}_{\ell_1}^{\rho_1} \dashv \Gamma' \quad \Gamma' \vdash e_2 : \mathtt{flip}^{\rho_2} \dashv \Gamma'' \quad \rho_1 \sqsubset \rho_2}{\Gamma \vdash \mathtt{xor}(e_1, e_2) : \mathtt{flip}^{\rho_2} \dashv \Gamma''}$$

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

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

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

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

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

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

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

Let-Tup
$$\frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2 \dashv \Gamma' \quad \Gamma'^{+} = \Gamma' \uplus [x_1 \mapsto \tau_1, x_2 \mapsto \tau_2]$$

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

$$\begin{array}{lcl}
\text{step}_{\mathcal{M}}(N, \sigma, b_\ell) & \triangleq & \text{return}(\sigma, \texttt{bitv}_\ell(b)) \\
\text{step}_{\mathcal{M}}(N, \sigma, \texttt{flip}^\rho()) & \triangleq & \text{do } b \leftarrow \text{bit}(N+1) \text{ ; return}(\sigma, \texttt{flipv}(b)) \\
\text{step}_{\mathcal{M}}(N, \sigma, \texttt{cast}_\ell(\texttt{flipv}(b))) & \triangleq & \text{return}(\sigma, \texttt{bitv}_\ell(b)) \\
\text{step}_{\mathcal{M}}(N, \sigma, \texttt{mux}(\texttt{bitv}_{\ell_1}(b_1), \texttt{bitv}_{\ell_2}(b_2), \texttt{bitv}_{\ell_3}(b_3))) & \triangleq & \text{return}(\sigma, \langle \texttt{bitv}_\ell(\text{cond}(b_1, b_2, b_3)), \texttt{bitv}_\ell(\text{cond}(b_1, b_3, b_2))\rangle) \\
 & & \quad \ell \triangleq \ell_1 \sqcup \ell_2 \sqcup \ell_3 \\
\text{step}_{\mathcal{M}}(N, \sigma, \texttt{mux}(\texttt{bitv}_\ell(b_1), \texttt{flipv}(b_2), \texttt{flipv}(b_3))) & \triangleq & \text{return}(\sigma, \langle \texttt{flipv}(\text{cond}(b_1, b_2, b_3)), \texttt{flipv}(\text{cond}(b_1, b_3, b_2))\rangle) \\
\text{step}_{\mathcal{M}}(N, \sigma, \texttt{xor}(\texttt{bitv}_\ell(b_1), \texttt{flipv}(b_2))) & \triangleq & \text{return}(\sigma, \texttt{flipv}(b_1 \oplus b_2)) \\
\text{step}_{\mathcal{M}}(N, \sigma, \texttt{if}(\texttt{bitv}_\ell(b))\{e_1\}\{e_2\}) & \triangleq & \text{return}(\sigma, \text{cond}(b, e_1, e_2)) \\
\text{step}_{\mathcal{M}}(N, \sigma, \texttt{ref}(v)) & \triangleq & \text{return}(\sigma[\iota \mapsto v], \texttt{refv}(\iota)) \quad \textit{where } \iota \notin \text{dom}(\sigma) \\
\text{step}_{\mathcal{M}}(N, \sigma, \texttt{read}(\texttt{refv}(\iota))) & \triangleq & \text{return}(\sigma, \sigma(\iota)) \\
\text{step}_{\mathcal{M}}(N, \sigma, \texttt{write}(\texttt{refv}(\iota), v)) & \triangleq & \text{return}(\sigma[\iota \mapsto v], \sigma(\iota)) \\
\text{step}_{\mathcal{M}}(N, \sigma, \texttt{let } x = v \texttt{ in } e) & \triangleq & \text{return}(\sigma, [v/x]e) \\
\text{step}_{\mathcal{M}}(N, \sigma, \texttt{let } x_1, x_2 = \langle v_1, v_2\rangle \texttt{ in } e) & \triangleq & \text{return}(\sigma, [v_1/x_1][v_2/x_2]e) \\
\text{step}_{\mathcal{M}}(N, \sigma, (\underbrace{\texttt{fun}_y(x:\tau).\ e}_{v_1})(v_2)) & \triangleq & \text{return}(\sigma, [v_1/y][v_2/x]e) \\
\text{step}_{\mathcal{M}}(N, \sigma, E[e]) & \triangleq & \text{do } \sigma', e' \leftarrow \text{step}_{\mathcal{M}}(N, \sigma, e) \text{ ; return}(\sigma', E[e']) \\
\text{step}_{\mathcal{M}}(N, \sigma, v) & \triangleq & \text{return}(\sigma, v)
\end{array}$$

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

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

$$\begin{array}{l}
\widehat{\text{step}}_{\mathcal{M}}(N, \hat{\varsigma}) \triangleq \text{do } \varsigma \leftarrow \hat{\varsigma} \text{ ; step}_{\mathcal{M}}(N, \varsigma) \\
\widehat{\text{nstep}}_{\mathcal{M}}(N, \hat{\varsigma}) \triangleq \text{do } \varsigma \leftarrow \hat{\varsigma} \text{ ; nstep}_{\mathcal{M}}(N, \varsigma)
\end{array}
\qquad
\begin{array}{l}
\boxed{\widehat{\text{step}}_{\mathcal{M}} \in \mathbb{N} \times \mathcal{M}(\text{config}) \rightharpoonup \mathcal{M}(\text{config})} \\
\boxed{\widehat{\text{nstep}}_{\mathcal{M}} \in \mathbb{N} \times \mathcal{M}(\text{config}) \rightharpoonup \mathcal{M}(\text{trace})}
\end{array}$$

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

$$\begin{array}{lcl}
\underline{\text{step}}(N, \underline{\sigma}, b_\ell) & \triangleq & \text{return}(\underline{\sigma}, \texttt{bitv}_\ell(\text{return}(b))) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{flip}^\rho()) & \triangleq & \text{return}(\underline{\sigma}, \texttt{flipv}(\text{bit}(N+1))) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{cast}_S(\texttt{flipv}(\hat{b}))) & \triangleq & \text{return}(\underline{\sigma}, \texttt{bitv}_S(\hat{b})) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{cast}_P(\texttt{flipv}(\hat{b}))) & \triangleq & \text{do } b \leftarrow \hat{b} \text{ ; return}(\underline{\sigma}, \texttt{bitv}_P(\text{return}(b))) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{mux}(\texttt{bitv}_{\ell_1}(\hat{b}_1), \texttt{bitv}_{\ell_2}(\hat{b}_2), \texttt{bitv}_{\ell_3}(\hat{b}_3))) & \triangleq & \text{return}(\underline{\sigma}, \langle \texttt{bitv}_\ell(\widehat{\text{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)), \texttt{bitv}_\ell(\widehat{\text{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2))\rangle) \\
\quad \textit{where } \ell \triangleq \ell_1 \sqcup \ell_2 \sqcup \ell_3 & & \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{mux}(\texttt{bitv}_\ell(\hat{b}_1), \texttt{flipv}(\hat{b}_2), \texttt{flipv}(\hat{b}_3))) & \triangleq & \text{return}(\underline{\sigma}, \langle \texttt{flipv}(\widehat{\text{cond}}(\hat{b}_1, \hat{b}_2, \hat{b}_3)), \texttt{flipv}(\widehat{\text{cond}}(\hat{b}_1, \hat{b}_3, \hat{b}_2))\rangle) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{xor}(\texttt{bitv}_{\ell_1}(\hat{b}_1), \texttt{flipv}(\hat{b}_2))) & \triangleq & \text{return}(\underline{\sigma}, \texttt{flipv}(\hat{b}_1 \mathbin{\hat{\oplus}} \hat{b}_2)) \\
\underline{\text{step}}(N, \underline{\sigma}, \texttt{if}(\texttt{bitv}_\ell(\hat{b}))\{\underline{e}_1\}\{\underline{e}_2\}) & \triangleq & \text{do } b \leftarrow \hat{b} \text{ ; return}(\underline{\sigma}, \text{cond}(b, \underline{e}_1, \underline{e}_2))
\end{array}$$

$$\boxed{\mathrm{obs} \in (\mathrm{exp} \to \dot{\mathrm{exp}}) \uplus (\mathrm{store} \to \dot{\mathrm{store}}) \uplus (\mathrm{config} \to \dot{\mathrm{config}}) \uplus (\mathrm{trace} \to \dot{\mathrm{trace}})}$$

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

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

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

$$\boxed{\begin{array}{l}\widetilde{\mathrm{obs}} \in \mathcal{D}(\mathrm{trace}) \to \mathcal{D}(\dot{\mathrm{trace}}) \\ \widehat{\mathrm{obs}} \in \mathcal{I}(\mathrm{trace}) \to \mathcal{I}(\dot{\mathrm{trace}})\end{array}}$$

$$\widetilde{\mathrm{obs}}(\tilde{t}) \triangleq \mathrm{do}\ t \leftarrow \tilde{t}\ ;\ \mathrm{return}(\mathrm{obs}(t)) \qquad \widehat{\mathrm{obs}}(\hat{t}) \triangleq \mathrm{do}\ t \leftarrow \hat{t}\ ;\ \mathrm{return}(\mathrm{obs}(t))$$

Figure B.5: Adversary Observation

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

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

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

$$
\begin{array}{lcl}
\lceil \varnothing \rceil & \triangleq & \text{return}(\varnothing) \\
\lceil \{\iota \mapsto \underline{v}\} \uplus \underline{\sigma} \rceil & \triangleq & \text{do } v \leftarrow \lceil \underline{v} \rceil \ ; \sigma \leftarrow \lceil \underline{\sigma} \rceil \ ; \text{return}(\{\iota \mapsto v\} \uplus \sigma)
\end{array}
$$

$\boxed{\underline{\text{config}} \to \mathcal{I}(\text{config})}$

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

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

$$
\begin{array}{lcl}
\lceil \epsilon \rceil & \triangleq & \text{return}(\epsilon) \\
\lceil \underline{t} \cdot \underline{\varsigma} \rceil & \triangleq & \text{do } t \leftarrow \underline{t} \ ; \varsigma \leftarrow \underline{\varsigma} \ ; \text{return}(t \cdot \varsigma)
\end{array}
$$

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

VarU

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

VarA

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

Bit

$$\frac{}{\Sigma, \Gamma \vdash b_\ell : \mathtt{bit}^{\perp}_{\ell} \;;\; \Gamma}$$

Flip

$$\frac{\rho \neq \perp}{\Sigma, \Gamma \vdash \mathtt{flip}^{\rho}() : \mathtt{flip}^{\rho} \;;\; \Gamma}$$

Cast-S

$$\frac{\Sigma, \Gamma \vdash \underline{v} : \mathtt{flip}^{\rho} \;;\; \_}{\Sigma, \Gamma \vdash \mathtt{cast}_{\mathrm{S}}(\underline{v}) : \mathtt{bit}^{\rho}_{\mathrm{S}} \;;\; \Gamma}$$

Cast-P

$$\frac{\Sigma, \Gamma \vdash \underline{v} : \mathtt{flip}^{\rho} \;;\; \Gamma'}{\Sigma, \Gamma \vdash \mathtt{cast}_{\mathrm{P}}(\underline{v}) : \mathtt{bit}^{\perp}_{\mathrm{P}} \;;\; \Gamma'}$$

If

$$\frac{\begin{array}{llll} \Sigma, \Gamma & \vdash & \underline{e}_1 & : \mathtt{bit}^{\perp}_{\mathrm{P}} \;;\; \Gamma' \\ \Sigma, \Gamma' & \vdash & \underline{e}_2 & : \tau \;;\; \Gamma''_1 \\ \Sigma, \Gamma' & \vdash & \underline{e}_3 & : \tau \;;\; \Gamma''_2 \end{array}}{\Sigma, \Gamma \vdash \mathtt{if}(\underline{e}_1)\{\underline{e}_2\}\{\underline{e}_3\} : \tau \;;\; \Gamma''_1 \sqcup \Gamma''_2}$$

Mux-Bit

$$\frac{\begin{array}{llllll} \Sigma, \Gamma & \vdash & \underline{e}_1 : \mathtt{bit}^{\rho_1}_{\ell_1} \;;\; \Gamma' & & & \\ \Sigma, \Gamma' & \vdash & \underline{e}_2 : \mathtt{bit}^{\rho_2}_{\ell_2} \;;\; \Gamma'' & \ell & = & \ell_1 \sqcup \ell_2 \sqcup \ell_3 \\ \Sigma, \Gamma'' & \vdash & \underline{e}_3 : \mathtt{bit}^{\rho_3}_{\ell_3} \;;\; \Gamma''' & \rho & = & \rho_1 \sqcup \rho_2 \sqcup \rho_3 \end{array}}{\Sigma, \Gamma \vdash \mathtt{mux}(\underline{e}_1, \underline{e}_2, \underline{e}_3) : \mathtt{bit}^{\rho}_{\ell} \times \mathtt{bit}^{\rho}_{\ell} \;;\; \Gamma'''}$$

Mux-Flip

$$\frac{\begin{array}{llllll} \Sigma, \Gamma & \vdash & \underline{e}_1 : \mathtt{bit}^{\rho_1}_{\ell_1} \;;\; \Gamma' & \rho_1 & \sqsubset & \rho_2 \\ \Sigma, \Gamma' & \vdash & \underline{e}_2 : \mathtt{flip}^{\rho_2} \;;\; \Gamma'' & \rho_1 & \sqsubset & \rho_3 \\ \Sigma, \Gamma'' & \vdash & \underline{e}_3 : \mathtt{flip}^{\rho_3} \;;\; \Gamma''' & \rho = \rho_2 \sqcap \rho_3 & & \end{array}}{\Sigma, \Gamma \vdash \mathtt{mux}(\underline{e}_1, \underline{e}_2, \underline{e}_3) : \mathtt{flip}^{\rho} \times \mathtt{flip}^{\rho} \;;\; \Gamma'''}$$

Xor-Flip

$$\frac{\begin{array}{llllll} \Sigma, \Gamma & \vdash & \underline{e}_1 : \mathtt{bit}^{\rho_1}_{\ell_1} \;;\; \Gamma' & \rho_1 & \sqsubset & \rho_2 \\ \Sigma, \Gamma' & \vdash & \underline{e}_2 : \mathtt{flip}^{\rho_2} \;;\; \Gamma'' & \rho = \rho_2 & & \end{array}}{\Sigma, \Gamma \vdash \mathtt{xor}(\underline{e}_1, \underline{e}_2) : \mathtt{flip}^{\rho} \;;\; \Gamma''}$$

Ref

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

Read

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

Write

$$\frac{\begin{array}{llll} \Sigma, \Gamma & \vdash & \underline{e}_1 : \mathtt{ref}(\tau) \;;\; \Gamma' \\ \Sigma, \Gamma' & \vdash & \underline{e}_2 : \tau \;;\; \Gamma'' \end{array}}{\Sigma, \Gamma \vdash \mathtt{write}(\underline{e}_1, \underline{e}_2) : \tau \;;\; \Gamma''}$$

Tup

$$\frac{\begin{array}{l} \Sigma, \Gamma \vdash \underline{e}_1 : \tau_1 \;;\; \Gamma' \\ \Sigma, \Gamma' \vdash \underline{e}_2 : \tau_2 \;;\; \Gamma'' \end{array}}{\Sigma, \Gamma \vdash \langle \underline{e}_1, \underline{e}_2 \rangle : \tau_1 \times \tau_2 \;;\; \Gamma''}$$

Fun

$$\frac{\begin{array}{llll} & & \Gamma^{+} & = \Gamma \uplus [x \mapsto \tau_1, y \mapsto (\tau_1 \to \tau_2)] \\ \Sigma, \Gamma^{+} & \vdash \underline{e} : \tau_2 \;;\; \Gamma^{+\prime} & \Gamma^{+\prime} & = \Gamma \uplus [x \mapsto \_, y \mapsto \_] \end{array}}{\Sigma, \Gamma \vdash \mathtt{fun}_y(x : \tau_1).\ \underline{e} : \tau_1 \to \tau_2 \;;\; \Gamma}$$

App

$$\frac{\begin{array}{llll} \Sigma, \Gamma & \vdash & \underline{e}_1 : \tau_1 \to \tau_2 \;;\; \Gamma' \\ \Sigma, \Gamma' & \vdash & \underline{e}_2 : \tau_1 \;;\; \Gamma'' \end{array}}{\Gamma \vdash \underline{e}_1(\underline{e}_2) : \tau_2 \;;\; \Gamma''}$$

Let

$$\frac{\begin{array}{llllll} \Sigma, \Gamma & \vdash & \underline{e}_1 : \tau_1 \;;\; \Gamma' & \Gamma'^{+} & = & \Gamma' \uplus [x \mapsto \tau_1] \\ \Sigma, \Gamma'^{+} & \vdash & \underline{e}_2 : \tau_2 \;;\; \Gamma''^{+} & \Gamma''^{+} & = & \Gamma'' \uplus [x \mapsto \_] \end{array}}{\Sigma, \Gamma \vdash \mathtt{let}\ x = \underline{e}_1\ \mathtt{in}\ \underline{e}_2 : \tau_2 \;;\; \Gamma''}$$

Let-Tup

$$\frac{\begin{array}{llllll} \Sigma, \Gamma & \vdash & \underline{e}_1 : \tau_1 \times \tau_2 \;;\; \Gamma' & \Gamma'^{+} & = & \Gamma' \uplus [x_1 \mapsto \tau_1, x_2 \mapsto \tau_2] \\ \Sigma, \Gamma'^{+} & \vdash & \underline{e}_2 : \tau_3 \;;\; \Gamma''^{+} & \Gamma''^{+} & = & \Gamma'' \uplus [x_1 \mapsto \_, x_2 \mapsto \_] \end{array}}{\Sigma, \Gamma \vdash \mathtt{let}\ x_1, x_2 = \underline{e}_1\ \mathtt{in}\ \underline{e}_2 : \tau_3 \;;\; \Gamma''}$$

BitV

$$\frac{}{\Sigma, \Gamma \vdash \mathtt{bitv}_{\ell}(b) : \mathtt{bit}^{\rho}_{\ell} \;;\; \Gamma}$$

FlipV

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

LocV

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

$\boxed{\Sigma \vdash \sigma}$

Store-Empty

$$\frac{}{\Sigma \vdash \varnothing}$$

Store-Cons

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

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

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

$\boxed{\Psi, \Phi, \Sigma, \Gamma \vdash \underline{e} : \tau \; ; \; \Gamma, \Psi}$

VarU

$$\frac{\mathcal{K}(\Gamma(x)) = \mathtt{U} \qquad \Gamma(x) = \tau}{\Psi_c, \Phi, \Sigma, \Gamma \vdash x : \tau \; ; \; \Gamma, \varnothing, \varnothing}$$

VarA

$$\frac{\mathcal{K}(\Gamma(x)) = \mathtt{A} \qquad \Gamma(x) = \tau}{\Psi_c, \Phi, \Sigma, \Gamma \vdash x : \tau \; ; \; \Gamma[x \mapsto \bullet], \varnothing, \varnothing}$$

Bit

$$\frac{}{\Psi_c, \Phi, \Sigma, \Gamma \vdash b_\ell : \mathtt{bit}_\ell^\perp \; ; \; \Gamma, \varnothing, \varnothing}$$

Flip

$$\frac{\rho \neq \bot}{\Psi_c, \Phi, \Sigma, \Gamma \vdash \mathtt{flip}^\rho() : \mathtt{flip}^\rho \; ; \; \Gamma, \varnothing, \varnothing}$$

Cast-S

$$\frac{\Psi_c, \Phi, \Sigma, \Gamma \vdash \underline{v} : \mathtt{flip}^\rho \; ; \; \_, \{\hat{b}\}, \varnothing}{\Psi_c, \Phi, \Sigma, \Gamma \vdash \mathtt{cast}_\mathrm{S}(\underline{v}) : \mathtt{bit}_\mathrm{S}^\rho \; ; \; \Gamma, \varnothing, \{\rho \mapsto \{\hat{b}\}\}}$$

Cast-P

$$\frac{\Psi_c, \Phi, \Sigma, \Gamma \vdash \underline{v} : \mathtt{flip}^\rho \; ; \; \Gamma', \Psi}{\Psi_c, \Phi, \Sigma, \Gamma \vdash \mathtt{cast}_\mathrm{P}(\underline{v}) : \mathtt{bit}_\mathrm{P}^\perp \; ; \; \Gamma', \Psi}$$

If

$$\frac{\begin{array}{lcl} \Psi_c \uplus \Psi_2 \uplus \Psi_3, \Phi, \Sigma, \Gamma & \vdash & \underline{e}_1 : \mathtt{bit}_\mathrm{P}^\perp \; ; \; \Gamma', \Psi_1 \\ \Psi_c \uplus \Psi_1 \uplus \Psi_3, \Phi, \Sigma, \Gamma' & \vdash & \underline{e}_2 : \tau \; ; \; \Gamma_1'', \Psi_2 \\ \Psi_c \uplus \Psi_1 \uplus \Psi_2, \Phi, \Sigma, \Gamma' & \vdash & \underline{e}_3 : \tau \; ; \; \Gamma_2'', \Psi_3 \end{array}}{\Psi_c, \Phi, \Sigma, \Gamma \vdash \mathtt{if}(\underline{e}_1)\{\underline{e}_2\}\{\underline{e}_3\} : \tau \; ; \; \Gamma_1'' \sqcup \Gamma_2'', \Psi_1 \uplus \Psi_2 \uplus \Psi_3}$$

Mux-Bit

$$\frac{\begin{array}{lcl} \Psi_c \uplus \Psi_2 \uplus \Psi_3, \Phi, \Sigma, \Gamma & \vdash & \underline{e}_1 : \mathtt{bit}_{\ell_1}^{\rho_1} \; ; \; \Gamma', \Psi_1 \\ \Psi_c \uplus \Psi_1 \uplus \Psi_3, \Phi, \Sigma, \Gamma' & \vdash & \underline{e}_2 : \mathtt{bit}_{\ell_2}^{\rho_2} \; ; \; \Gamma'', \Psi_2 \quad \ell = \ell_1 \sqcup \ell_2 \sqcup \ell_3 \\ \Psi_c \uplus \Psi_1 \uplus \Psi_2, \Phi, \Sigma, \Gamma'' & \vdash & \underline{e}_3 : \mathtt{bit}_{\ell_3}^{\rho_3} \; ; \; \Gamma''', \Psi_3 \quad \rho = \rho_1 \sqcup \rho_2 \sqcup \rho_3 \end{array}}{\Psi_c, \Phi, \Sigma, \Gamma \vdash \mathtt{mux}(\underline{e}_1, \underline{e}_2, \underline{e}_3) : \mathtt{bit}_\ell^\rho \times \mathtt{bit}_\ell^\rho \; ; \; \Gamma''', \Psi_1 \uplus \Psi_2 \uplus \Psi_3}$$

Mux-Flip

$$\frac{\begin{array}{lcl} \Psi_c \uplus \Psi_2 \uplus \Psi_3, \Phi, \Sigma, \Gamma & \vdash & \underline{e}_1 : \mathtt{bit}_{\ell_1}^{\rho_1} \; ; \; \Gamma', \Psi_1 \quad \rho_1 \sqsubset \rho_2 \\ \Psi_c \uplus \Psi_1 \uplus \Psi_3, \Phi, \Sigma, \Gamma' & \vdash & \underline{e}_2 : \mathtt{flip}^{\rho_2} \; ; \; \Gamma'', \Psi_2 \quad \rho_1 \sqsubset \rho_3 \\ \Psi_c \uplus \Psi_1 \uplus \Psi_2, \Phi, \Sigma, \Gamma'' & \vdash & \underline{e}_3 : \mathtt{flip}^{\rho_3} \; ; \; \Gamma''', \Psi_3 \quad \rho = \rho_2 \sqcap \rho_3 \end{array}}{\Psi_c, \Phi, \Sigma, \Gamma \vdash \mathtt{mux}(\underline{e}_1, \underline{e}_2, \underline{e}_3) : \mathtt{flip}^\rho \times \mathtt{flip}^\rho \; ; \; \Gamma''', \Psi_1 \uplus \Psi_2 \uplus \Psi_3}$$

Xor-Flip

$$\frac{\begin{array}{lcl} \Psi_c \uplus \Psi_2, \Phi, \Sigma, \Gamma & \vdash & \underline{e}_1 : \mathtt{bit}_{\ell_1}^{\rho_1} \; ; \; \Gamma', \Psi_1 \quad \rho_1 \sqsubset \rho_2 \\ \Psi_c \uplus \Psi_1, \Phi, \Sigma, \Gamma' & \vdash & \underline{e}_2 : \mathtt{flip}^{\rho_2} \; ; \; \Gamma'', \Psi_2 \quad \rho = \rho_2 \end{array}}{\Psi_c, \Phi, \Sigma, \Gamma \vdash \mathtt{xor}(\underline{e}_1, \underline{e}_2) : \mathtt{flip}^\rho \; ; \; \Gamma'', \Psi_1 \uplus \Psi_2}$$

Ref

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

Read

$$\frac{\mathcal{K}(\tau) = \mathtt{U} \qquad \Psi_c, \Phi, \Sigma, \Gamma \vdash \underline{e} : \mathtt{ref}(\tau) \; ; \; \Gamma', \Psi}{\Psi_c, \Phi, \Sigma, \Gamma \vdash \mathtt{read}(\underline{e}) : \tau \; ; \; \Gamma', \Psi}$$

Write

$$\frac{\begin{array}{lcl} \Psi_c \uplus \Psi_2, \Phi, \Sigma, \Gamma & \vdash & \underline{e}_1 : \mathtt{ref}(\tau) \; ; \; \Gamma', \Psi_1 \\ \Psi_c \uplus \Psi_1, \Phi, \Sigma, \Gamma' & \vdash & \underline{e}_2 : \tau \; ; \; \Gamma'', \Psi_2 \end{array}}{\Psi_c, \Phi, \Sigma, \Gamma \vdash \mathtt{write}(\underline{e}_1, \underline{e}_2) : \tau \; ; \; \Gamma'', \Psi_1 \uplus \Psi_2}$$

Tup

$$\frac{\begin{array}{c} \Psi_c \uplus \Psi_2, \Phi, \Sigma, \Gamma \vdash \underline{e}_1 : \tau_1 \; ; \; \Gamma', \Psi_1 \\ \Psi_c \uplus \Psi_1, \Phi, \Sigma, \Gamma' \vdash \underline{e}_2 : \tau_2 \; ; \; \Gamma'', \Psi_2 \end{array}}{\Psi_c, \Phi, \Sigma, \Gamma \vdash \langle \underline{e}_1, \underline{e}_2 \rangle : \tau_1 \times \tau_2 \; ; \; \Gamma'', \Psi_1 \uplus \Psi_2}$$

Fun

$$\frac{\begin{array}{c} \Gamma^+ = \Gamma \uplus [x \mapsto \tau_1, y \mapsto (\tau_1 \to \tau_2)] \\ \Gamma^{+\prime} = \Gamma \uplus [x \mapsto \_, y \mapsto \_] \\ \Psi_c, \Phi, \Sigma, \Gamma^+ \vdash \underline{e} : \tau_2 \; ; \; \Gamma^{+\prime}, \Psi \end{array}}{\Psi_c, \Phi, \Sigma, \Gamma \vdash \mathtt{fun}_y(x : \tau_1).\ \underline{e} : \tau_1 \to \tau_2 \; ; \; \Gamma, \Psi}$$

App

$$\frac{\begin{array}{lcl} \Psi_c \uplus \Psi_2, \Phi, \Sigma, \Gamma & \vdash & \underline{e}_1 : \tau_1 \to \tau_2 \; ; \; \Gamma', \Psi_1 \\ \Psi_c \uplus \Psi_1, \Phi, \Sigma, \Gamma' & \vdash & \underline{e}_2 : \tau_1 \; ; \; \Gamma'', \Psi_2 \end{array}}{\Gamma \vdash \underline{e}_1(\underline{e}_2) : \tau_2 \; ; \; \Gamma''}$$

Let

$$\frac{\begin{array}{lcl} \Psi_c \uplus \Psi_2, \Phi, \Sigma, \Gamma & \vdash & \underline{e}_1 : \tau_1 \; ; \; \Gamma', \Psi_1 \quad \Gamma'^{+} = \Gamma' \uplus [x \mapsto \tau_1] \\ \Psi_c \uplus \Psi_1, \Phi, \Sigma, \Gamma'^{+} & \vdash & \underline{e}_2 : \tau_2 \; ; \; \Gamma''^{+}, \Psi_2 \quad \Gamma''^{+} = \Gamma'' \uplus [x \mapsto \_] \end{array}}{\Psi_c \uplus \Psi_1 \uplus \Psi_2, \Phi, \Sigma, \Gamma \vdash \mathtt{let}\ x = \underline{e}_1\ \mathtt{in}\ \underline{e}_2 : \tau_2 \; ; \; \Gamma'', \Psi_1 \uplus \Psi_2}$$

$\boxed{\Psi, \Phi \vdash \hat{b} : \texttt{flip}^{\rho}}$

Flip-Value

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

$\boxed{\Psi, \Phi, \Sigma, \Gamma \vdash v : \tau \,;\, \Gamma, \Psi}$

BitV-P

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

BitV-S

$$\frac{}{\Psi_c, \Phi, \Sigma, \Gamma \vdash \texttt{bitv}_S(\hat{b}) : \texttt{bit}_{\ell}^{\rho} \,;\, \Gamma, \varnothing, \{\rho \mapsto \{\hat{b}\}\}}$$

FlipV

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

LocV

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

$\boxed{\Psi, \Phi, \Sigma \vdash \sigma \,;\, \Psi}$

Store-Empty

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

Store-Cons

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

$\boxed{\Psi, \Phi, \Sigma \vdash \varsigma : \tau, \Psi}$

Config

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

$\underline{e}_1 \sim \underline{e}_2$

BitV-P

$$\frac{}{\texttt{bitv}_P(\hat{b}) \sim \texttt{bitv}_P(\hat{b})}$$

BitV-S

$$\frac{}{\texttt{bitv}_S(\hat{b}) \sim \texttt{bitv}_S(\hat{b}')}$$

FlipV

$$\frac{}{\texttt{flipv}(\hat{b}) \sim \texttt{flipv}(\hat{b}')}$$

LocV

$$\frac{}{\texttt{locv}(\iota) \sim \texttt{locv}(\iota)}$$

Var

$$\frac{}{x \sim x}$$

BitP

$$\frac{}{b_P \sim b_P}$$

BitS

$$\frac{}{b_S \sim b'_S}$$

Flip

$$\frac{}{\texttt{flip}^{\rho}() \sim \texttt{flip}^{\rho}()}$$

Cast

$$\frac{\underline{v} \sim \underline{v}'}{\texttt{cast}_{\ell}(\underline{v}) \sim \texttt{cast}_{\ell}(\underline{v}')}$$

If

$$\frac{\underline{e}_1 \sim \underline{e}'_1 \qquad \underline{e}_2 \sim \underline{e}'_2 \qquad \underline{e}_3 \sim \underline{e}'_3}{\texttt{if}(\underline{e}_1)\{\underline{e}_2\}\{\underline{e}_3\} \sim \texttt{if}(\underline{e}'_1)\{\underline{e}'_2\}\{\underline{e}'_3\}}$$

Mux

$$\frac{\underline{e}_1 \sim \underline{e}'_1 \qquad \underline{e}_2 \sim \underline{e}'_2 \qquad \underline{e}_3 \sim \underline{e}'_3}{\texttt{mux}(\underline{e}_1, \underline{e}_2, \underline{e}_3) \sim \texttt{mux}(\underline{e}'_1, \underline{e}'_2, \underline{e}'_3)}$$

Xor

$$\frac{\underline{e}_1 \sim \underline{e}'_1 \qquad \underline{e}_2 \sim \underline{e}'_2}{\texttt{xor}(\underline{e}_1, \underline{e}_2) \sim \texttt{xor}(\underline{e}'_1, \underline{e}'_2)}$$

Ref

$$\frac{\underline{e}_1 \sim \underline{e}'_1}{\texttt{ref}(\underline{e}_1) \sim \texttt{ref}(\underline{e}'_1)}$$

Read

$$\frac{\underline{e} \sim \underline{e}'}{\texttt{read}(\underline{e}) \sim \texttt{read}(\underline{e}')}$$

Write

$$\frac{\underline{e}_1 \sim \underline{e}'_1 \qquad \underline{e}_2 \sim \underline{e}'_2}{\texttt{write}(\underline{e}_1, \underline{e}_2) \sim \texttt{write}(\underline{e}'_1, \underline{e}'_2)}$$

Tup

$$\frac{\underline{e}_1 \sim \underline{e}'_1 \qquad \underline{e}_2 \sim \underline{e}'_2}{\langle \underline{e}_1, \underline{e}_2 \rangle \sim \langle \underline{e}'_1, \underline{e}'_2 \rangle}$$

Fun

$$\frac{\underline{e} \sim \underline{e}'}{\texttt{fun}_y(x : \tau).\ \underline{e} \sim \texttt{fun}_y(x : \tau).\ \underline{e}'}$$

App

$$\frac{\underline{e}_1 \sim \underline{e}'_1 \qquad \underline{e}_2 \sim \underline{e}'_2}{\underline{e}_1(\underline{e}_2) \sim \underline{e}'_1(\underline{e}'_2)}$$

Let

$$\frac{\underline{e}_1 \sim \underline{e}'_1 \qquad \underline{e}_2 \sim \underline{e}'_2}{\texttt{let}\ x = \underline{e}_1\ \texttt{in}\ \underline{e}_2 \sim \texttt{let}\ x = \underline{e}'_1\ \texttt{in}\ \underline{e}'_2}$$

Let-Tup

$$\frac{\underline{e}_1 \sim \underline{e}'_1 \qquad \underline{e}_2 \sim \underline{e}'_2}{\texttt{let}\ x, y = \underline{e}_1\ \texttt{in}\ \underline{e}_2 \sim \texttt{let}\ x, y = \underline{e}'_1\ \texttt{in}\ \underline{e}'_2}$$

Figure B.10: Low Equivalence Relation

Appendix C

Bounding Information Leakage: Evacuation Scenario and Proofs

C.1 Query Code

The following is the query code of the example developed in Section 5.1.2. Here, `s_x` and `s_y` represent a ship's secret location. The variables `l1_x`, `l1_y`, `l2_x`, `l2_y`, and `d` are inputs to the query. The first pair represents position L_1, the second pair represents the position L_2, and the last is the distance threshold, set to 4. We assume for the example that L_1 and L_2 have the same y coordinate, and their x coordinates differ by 6 units.

We express the query in the language of Figure 5.4 basically as follows:

```
d_l1 := |s_x - l1_x| + |s_y - l1_y|;
d_l2 := |s_x - l2_x| + |s_y - l2_y|;
if (d_l1 <= d || d_l2 <= d) then
  out := true // assume this result
else
  out := false
```

The variable out is the result of the query. We simplify the code by assuming the absolute value function is built-in; we can implement this with a simple conditional. We run this query probabilistically under the assumption that s_x and s_y are uniformly distributed within the range given in Figure 5.1. We then condition the output on the assumption that out = true. When using intervals as the baseline of probabilistic polyhedra, this produces the result given in the upper right of Figure 5.3(b); when using convex polyhedra, the result is shown in the lower right of the figure. The use of sampling and concolic execution to augment the former is shown via arrows between the two.

C.2 Formal semantics

Here we defined the probabilistic semantics for the programming language given in Figure 5.4. The semantics of statement S, written $[\![S]\!]$, is a function of the form **Dist** $\to$ **Dist**, i.e., it is a function from distributions of states to distributions of states. We write $[\![S]\!]\delta = \delta'$ to say that the semantics of S maps input distribution δ to output distribution δ'.

Figure C.1 gives this denotational semantics along with definitions of relevant auxiliary operations. We write $[\![E]\!]\sigma$ to denote the (integer) result of evaluating expression E in σ, and $[\![B]\!]\sigma$ to denote the truth or falsehood of B in σ. The variables of a state σ, written $\mathit{domain}(\sigma)$, is defined by $\mathit{domain}(\sigma)$; sometimes we will refer to this set as just the *domain* of σ. We will also use the this notation for distributions; $\mathit{domain}(\delta) \stackrel{\text{def}}{=} \mathit{domain}(\mathit{domain}(\delta))$. We write lfp as the least fixed-point operator. The notation $\sum_{x\,:\,\phi} \rho$ can be read *ρ is the sum over all x such that formula ϕ is satisfied* (where x is bound in ρ and ϕ).

This semantics is standard. See Clarkson et al. [44] or Mardziel et al [115] for detailed explanations.

$$
\begin{array}{rcl}
[\![\mathsf{skip}]\!]\delta & = & \delta \\
[\![x \;:=\; E]\!]\delta & = & \delta\,[x \to E] \\
[\![\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2]\!]\delta & = & [\![S_1]\!](\delta \wedge B) + [\![S_2]\!](\delta \wedge \neg B) \\
[\![\mathsf{pif}\ q\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2]\!]\delta & = & [\![S_1]\!](q \cdot \delta) + [\![S_2]\!]((1-q) \cdot \delta) \\
[\![S_1\ ;\ S_2]\!]\delta & = & [\![S_2]\!]([\![S_1]\!]\delta) \\
[\![\mathsf{while}\ B\ \mathsf{do}\ S]\!] & = & \mathrm{lfp}\,[\lambda f :\ \mathbf{Dist} \to \mathbf{Dist}.\ \lambda\delta. \\
 & & \quad f\,([\![S]\!](\delta \wedge B)) + (\delta \wedge \neg B)]
\end{array}
$$

where

$$
\begin{array}{ll}
\delta\,[x \to E] & \stackrel{\text{def}}{=} \lambda\sigma.\ \sum_{\tau\,:\,\tau[x \to [\![E]\!]\tau] = \sigma} \delta(\tau) \\
\delta_1 + \delta_2 & \stackrel{\text{def}}{=} \lambda\sigma.\ \delta_1(\sigma) + \delta_2(\sigma) \\
\delta \wedge B & \stackrel{\text{def}}{=} \lambda\sigma.\ \mathbf{if}\ [\![B]\!]\sigma\ \mathbf{then}\ \delta(\sigma)\ \mathbf{else}\ 0 \\
p \cdot \delta & \stackrel{\text{def}}{=} \lambda\sigma.\ p \cdot \delta(\sigma) \\
\|\delta\| & \stackrel{\text{def}}{=} \sum_\sigma \delta(\sigma) \\
\mathrm{normal}(\delta) & \stackrel{\text{def}}{=} \frac{1}{\|\delta\|} \cdot \delta \\
\delta|B & \stackrel{\text{def}}{=} \mathrm{normal}(\delta \wedge B) \\
\delta_1 \times \delta_2 & \stackrel{\text{def}}{=} \lambda(\sigma_1, \sigma_2).\ \delta_1(\sigma_1) \cdot \delta_2(\sigma_2) \\
\dot{\sigma} & \stackrel{\text{def}}{=} \lambda\sigma_0.\ \mathbf{if}\ \sigma = \sigma_0\ \mathbf{then}\ 1\ \mathbf{else}\ 0 \\
\sigma \downharpoonright V & \stackrel{\text{def}}{=} \lambda x \in \mathbf{Var}_V.\ \sigma(x) \\
\delta \downharpoonright V & \stackrel{\text{def}}{=} \lambda\sigma_V \in \mathbf{State}_V.\ \sum_{\tau\,:\,\tau \downharpoonright V = \sigma_V} \delta(\tau) \\
\mathrm{f}_x(\delta) & \stackrel{\text{def}}{=} \delta \downharpoonright (\mathit{domain}(\delta) - \{x\}) \\
\mathit{support}(\delta) & \stackrel{\text{def}}{=} \{\sigma\ :\ \delta(\sigma) > 0\}
\end{array}
$$

Figure C.1: Distribution semantics

C.2.1 Proofs

Here we restate the soundness theorems for our techniques, and include their proofs.

Theorem C.2.2 (Sampling is Sound)**.**

If $\delta_0 \in \gamma_{\mathbb{P}}(P_0)$, $\langle\!\langle S \rangle\!\rangle\, P_0 = P$, *and* $[\![S]\!]\delta_0 = \delta$ *then*

$$\delta_T \in \gamma_{\mathbb{P}}(P_{T+}) \text{ with confidence } \omega$$

where

$$\delta_T \stackrel{\text{def}}{=} \delta \wedge (r = o) \lfloor T$$

$$P_T \stackrel{\text{def}}{=} P \wedge (r = o) \lfloor T$$

$$P_{T+} \stackrel{\text{def}}{=} P_T \text{ sampling revised with confidence } \omega.$$

Proof. Suppose we have some $\delta_0 \in \gamma_{\mathbb{P}}(P_0)$ whereby $[\![S]\!]\delta_0 = \delta$. We want to prove that $\delta_T \in \gamma_{\mathbb{P}}(P_{T+})$. Per Definition 5.2.2, this means we must show that

(1) $\mathit{support}(\delta_T) \subseteq \gamma_{\mathbb{P}}(C_{T+})$

(2) $\mathrm{s}^{\min}_{T+} \leq |\mathit{support}(\delta_T)| \leq \mathrm{s}^{\max}_{T+}$

(3) $\mathrm{m}^{\min}_{T+} \leq \|\delta_T\| \leq \mathrm{m}^{\max}_{T+}$

(4) $\forall \sigma \in \mathit{support}(\delta_T).\, \mathrm{p}^{\min}_{T+} \leq \delta_T(\sigma) \leq \mathrm{p}^{\max}_{T+}$

Our proof goes as follows. First, we know that $\delta_T \in \gamma_{\mathbb{P}}(C_T)$ by Theorem 5.2.1, Lemma 15 and Lemma 7 of Mardziel et al. By Definition 5.2.2, this means

(a) $\mathit{support}(\delta_T) \subseteq \gamma_{\mathbb{P}}(C_T)$

(b) $\mathrm{s}^{\min}_{T} \leq |\mathit{support}(\delta_T)| \leq \mathrm{s}^{\max}_{T}$

(c) $\mathrm{m}^{\min}_{T} \leq \|\delta_T\| \leq \mathrm{m}^{\max}_{T}$

(d) $\forall \sigma \in \mathit{support}(\delta_T).\, \mathrm{p}_T^{\min} \leq \delta_T(\sigma) \leq \mathrm{p}_T^{\max}$

So (1) and (4) follow directly from (a) and (d), since $\mathrm{p}_T^{\min} = \mathrm{p}_{T+}^{\min}$, $\mathrm{p}_T^{\max} = \mathrm{p}_{T+}^{\max}$, and $C_T = C_{T+}$.

To prove (2), we argue as follows. Let $p = \frac{|\mathit{support}(\delta_T)|}{\#(C_T)}$, which represents the probability that a randomly selected point from C_T is in $\mathit{support}(\delta_T)$. From the computed credible interval over the Beta distribution, we have that $p \in [p_L, p_U]$ with confidence ω. As such,

$$\begin{array}{rcl} p_L & \leq p \leq & p_U \\ p_L & \leq \frac{|\mathit{support}(\delta_T)|}{\#(C_T)} \leq & p_U \\ p_L \cdot \#(C_T) & \leq |\mathit{support}(\delta_T)| \leq & p_U \cdot \#(C_T) \\ \mathrm{s}_{T+}^{\min} & \leq |\mathit{support}(\delta_T)| \leq & \mathrm{s}_{T+}^{\max} \end{array}$$

which is the desired result.

To prove (3), first consider that if $\mathrm{m}_{T+}^{\min} = \mathrm{m}_T^{\min}$ then the first half of (3) follows from the first half of (c). Otherwise, we have that $\mathrm{m}_{T+}^{\min} = \mathrm{p}_T^{\min} \cdot \mathrm{s}_{T+}^{\min}$. Then we can reason the first half of (3) holds using the following reasoning:

$$\begin{array}{rcll} \mathrm{s}_{T+}^{\min} & \leq & |\mathit{support}(\delta_T)| & \text{by (2)} \\ \mathrm{p}_{T+}^{\min} \cdot \mathrm{s}_{T+}^{\min} & \leq & \mathrm{p}_{T+}^{\min} \cdot |\mathit{support}(\delta_T)| & \text{as } \mathrm{p}_{T+}^{\min} \text{ nonneg.} \\ \mathrm{m}_{T+}^{\min} & \leq & \mathrm{p}_{T+}^{\min} \cdot |\mathit{support}(\delta_T)| & \text{by def.} \\ & & = \Sigma_{\sigma \in \mathit{support}(\delta_T)} \mathrm{p}_{T+}^{\min} & \\ & & \leq \Sigma_{\sigma \in \mathit{support}(\delta_T)} \delta_T(\sigma) & \text{by (4)} \\ & & = \|\delta_T\| & \end{array}$$

We can prove the soundness of $\mathrm{m}_{T+}^{\max}$ (the other half of (3)) with similar reasoning. □

Theorem C.2.3 (Concolic Execution is Sound)**.**

If $\delta_0 \in \gamma_{\mathbb{P}}(P_0)$, $\langle\!\langle S \rangle\!\rangle\, P_0 = P$, *and* $[\![S]\!]\delta_0 = \delta$ *then*

$$\delta_T \in \gamma_{\mathbb{P}}(P_{T+})$$

where

$$\delta_T \stackrel{\text{def}}{=} \delta \wedge (r = o) \downharpoonright T$$

$$P_T \stackrel{\text{def}}{=} P \wedge (r = o) \downharpoonright T$$

$$P_{T+} \stackrel{\text{def}}{=} P_T \textit{ concolically revised.}$$

Proof. Our proof is quite similar to that of Theorem C.2.2. Once again we proceed to show the four elements of Definition 5.2.2, where (1) and (4) hold by construction. To prove (2) we are only concerned with the inequality $\mathrm{s}_T^{\min} \leq |\mathit{support}(\delta_T)|$, since $\mathrm{s}_{T+}^{\max} = \mathrm{s}_T^{\max}$. We know by the soundness of P_T that

$$\mathrm{s}_T^{\min} \leq |\mathit{support}(\delta_T)|$$

From the definition of concolic execution we have $\{\sigma \mid \sigma \in C_T \wedge \sigma \models \pi\} \subseteq \{\sigma \mid \delta_T(\sigma) > 0\}$. Notice that this is just saying that the concolic execution is a valid under-approximation for the support. From this we know that:

$$\begin{aligned}
|\{\sigma \mid \sigma \in C_T \wedge \sigma \models \pi\}| &\leq |\{\sigma \mid \delta_T(\sigma) > 0\}| \\
\#(C_T \sqcap (\bigsqcup_i C_i)) &\leq |\{\sigma \mid \delta_T(\sigma) > 0\}| \\
\#(C_T \sqcap (\bigsqcup_i C_i)) &\leq |\mathit{support}(\delta_T)| \\
\mathrm{s}_{T+}^{\min} &\leq |\mathit{support}(\delta_T)|
\end{aligned}$$

Given (2), our proof of (3) proceeds similarly to Theorem C.2.2. □

Theorem C.2.4 (Concolic and Sampling Composition is Sound)**.**

If $\delta_0 \in \gamma_{\mathbb{P}}(P_0)$, $\langle\langle S \rangle\rangle\, P_0 = P$, *and* $[\![S]\!]\delta_0 = \delta$ *then*

$$\delta_T \in \gamma_{\mathbb{P}}(P_{T+})$$

where

$$\delta_T \stackrel{\text{def}}{=} \delta \wedge (r = o) \downharpoonleft T$$

$$P_T \stackrel{\text{def}}{=} P \wedge (r = o) \downharpoonleft T$$

$$P_{T+} \stackrel{\text{def}}{=} P_T \textit{ sampling-and-concolically revised with confidence } \omega$$

Proof. Our proof is quite similar to that of Theorem 2. Once again we proceed to show the four elements of Definition 2, where (1) and (4) hold by construction. To prove (2), consider the set $|\mathit{support}(\delta_T)|$. For any state $\sigma_T \in \mathit{support}(\delta_T)$ it must be the case that either $\sigma_T \in C$ or $\sigma_T \in C_T \setminus C$. This is because $C \subseteq C_T$. Thus, we have

$$|\mathit{support}(\delta_T)| = |\mathit{support}(C_T \setminus C)| + |\mathit{support}(C)|$$

Additionally, since C represents a sound under-approximation of the support, we know that

$$|\mathit{support}(C)| = \#(C)$$

So, $p = \frac{|\mathit{support}(C_T \setminus C)|}{\#(C_T \setminus C)}$ represents the probability that a point in the region surrounding C is in the support of δ_T. Note that our procedure implements a uniform, random sample over this region. Thus, from the computed credible interval over the

Beta distribution, we have that $p \in [p_L, p_U]$ with confidence ω. As such,

$$\begin{aligned}
p_L &\leq p \leq && p_U \\
p_L &\leq \tfrac{|support(C_T \setminus C)|}{\#(C_T \setminus C)} \leq && p_U \\
p_L \cdot \#(C_T \setminus C) &\leq |support(C_T \setminus C)| \leq && p_U \cdot \#(C_T \setminus C) \\
p_L \cdot \#(C_T \setminus C) + |support(C)| &\leq |support(C_T \setminus C)| + |support(C)| \leq && p_U \cdot \#(C_T \setminus C) + |support(C)| \\
p_L \cdot \#(C_T \setminus C) + |support(C)| &\leq |support(\delta_T)| \leq && p_U \cdot \#(C_T \setminus C) + |support(C)| \\
p_L \cdot \#(C_T \setminus C) + \#(C) &\leq |support(\delta_T)| \leq && p_U \cdot \#(C_T \setminus C) + \#(C) \\
p_L \cdot (\#(C_T) - \#(C)) + \#(C) &\leq |support(\delta_T)| \leq && p_U \cdot (\#(C_T) - \#(C)) + \#(C) \\
\mathrm{s}_{T+}^{\min} &\leq |support(\delta_T)| \leq && \mathrm{s}_{T+}^{\max}
\end{aligned}$$

which is the desired result.

Given (2), our proof of (3) proceeds similarly to Theorem C.2.2 and C.2.3. □

Bibliography

[1] Martín Abadi et al. "A Core Calculus of Dependency". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, pp. 147–160. ISBN: 1581130953. DOI: 10.1145/292540.292555. URL: https://doi.org/10.1145/292540.292555.

[2] Coşku Acay et al. "Viaduct: An Extensible, Optimizing Compiler for Secure Distributed Programs". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 740–755. ISBN: 9781450383912. DOI: 10.1145/3453483.3454074. URL: https://doi.org/10.1145/3453483.3454074.

[3] Johan Agat. "Transforming out Timing Leaks". In: *POPL*. 2000.

[4] Mário S. Alvim et al. "Measuring Information Leakage Using Generalized Gain Functions". In: *Proc. IEEE Computer Security Foundations Symposium (CSF)*. 2012.

[5] Abdelrahaman Aly et al. *SCALE-MAMBA*. https://homes.esat.kuleuven.be/ nsmart/SCALE/. 2019.

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

[7] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999.

[8] Michael Backes, Boris Köpf, and Andrey Rybalchenko. "Automatic Discovery and Quantification of Information Leaks". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2009.

[9] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. "Widening operators for powerset domains". In: *International Journal on Software Tools for Tech. Transfer* 8.4 (2006), pp. 449–466.

[10] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. "The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems". In: *Sci. Comput. Program.* 72 (June 2008).

[11] Roberto Bagnara, Enric Rodríguez-Carbonell, and Enea Zaffanella. "Generation of basic semi-algebraic invariants using convex polyhedra". In: *SAS*. 2005.

[12] Henry G. Baker. "Lively Linear Lisp: "Look Ma, No Garbage!";" in: *SIGPLAN Not.* 27.8 (Aug. 1992), pp. 89–98. ISSN: 0362-1340. DOI: 10.1145/142137.142162. URL: http://doi.acm.org/10.1145/142137.142162.

[13] Marshall Ball, Tal Malkin, and Mike Rosulek. "Garbling Gadgets for Boolean and Arithmetic Circuits". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 565–577. ISBN: 9781450341394. DOI: 10.1145/2976749.2978410. URL: https://doi.org/10.1145/2976749.2978410.

[14] Tyler Barker. "A Monad for Randomized Algorithms". In: *Electronic Notes in Theoretical Computer Science* 325 (2016). The Thirty-second Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXII), pp. 47–62. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2016.09.031. URL: http://www.sciencedirect.com/science/article/pii/S1571066116300780.

[15] Barry Schoenmakers. *MPyC: Secure Multiparty Computation in Python*. July 25, 2022. URL: https://github.com/lschoe/mpyc.

[16] Gilles Barthe, Justin Hsu, and Kevin Liao. "A Probabilistic Separation Logic". In: *PACMPL* 4.POPL (2020).

[17] Gilles Barthe et al. "Security of multithreaded programs by compilation". In: *ACM Transactions on Information and System Security (TISSEC)* 13.3 (2010), p. 21.

[18] Gilles Barthe et al. "Probabilistic Relational Reasoning for Differential Privacy". In: *ACM Trans. Program. Lang. Syst.* 35.3 (2013), 9:1–9:49.

[19] Gilles Barthe et al. "Probabilistic Relational Verification for Cryptographic Implementations". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: ACM, 2014, pp. 193–205. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535847. URL: http://doi.acm.org/10.1145/2535838.2535847.

[20] Gilles Barthe et al. "Higher-order approximate relational refinement types for mechanism design and differential privacy". In: *ACM SIGPLAN Notices*. Vol. 50. 1. ACM. 2015, pp. 55–68.

[21] Gilles Barthe et al. "Coupling Proofs Are Probabilistic Product Programs". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, 2017, pp. 161–174. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009896. URL: http://doi.acm.org/10.1145/3009837.3009896.

[22] Gilles Barthe et al. "Proving uniformity and independence by self-composition and coupling". In: *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. Ed. by Thomas Eiter and David Sands. Vol. 46. EPiC Series in Computing. EasyChair, 2017, pp. 385–403. DOI: 10.29007/vz48. URL: https://easychair.org/publications/paper/L9T5.

[23] Gilles Barthe et al. "An Assertion-Based Program Logic for Probabilistic Programs". In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Cham: Springer International Publishing, 2018, pp. 117–144. ISBN: 978-3-319-89884-1.

[24] D. Beaver, S. Micali, and P. Rogaway. "The Round Complexity of Secure Protocols". In: *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*. STOC '90. Baltimore, Maryland, USA: Association for Computing Machinery, 1990, pp. 503–513. ISBN: 0897913612. DOI: 10.1145/100216.100287. URL: https://doi.org/10.1145/100216.100287.

[25] Donald Beaver. "Efficient Multiparty Protocols Using Circuit Randomization". In: *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*. Vol. 576. Lecture Notes in Computer Science. Springer, 1991, pp. 420–432. DOI: 10.1007/3-540-46766-1_34.

[26] Donald Beaver. "Correlated Pseudorandomness and the Complexity of Private Computations". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 479–488. ISBN: 0897917855. DOI: 10.1145/237814.237996. URL: https://doi.org/10.1145/237814.237996.

[27] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract)". In: *20th Annual ACM Symposium on Theory of Computing*. Chicago, IL, USA: ACM Press, May 1988, pp. 1–10. DOI: 10.1145/62212.62213.

[28] Frédéric Besson, Nataliia Bielova, and Thomas Jensen. "Browser Randomisation against Fingerprinting: A Quantitative Information Flow Approach". In: *NordSec*. 2014.

[29] Ken Biba. "Integrity Considerations for Secure Computer Systems". In: (Apr. 1977), p. 68.

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

[31] Dan Bogdanov, Sven Laur, and Jan Willemson. "Sharemind: A Framework for Fast Privacy-Preserving Computations". In: *ESORICS*. 2008.

[32] Dan Boneh et al. *Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs*. Cryptology ePrint Archive, Paper 2019/188. https://eprint.iacr.org/2019/188. 2019. URL: https://eprint.iacr.org/2019/188.

[33] Johannes Borgström et al. “Measure transformer semantics for Bayesian machine learning”. In: *Proceedings of the European Symposium on Programming (ESOP)*. 2011. ISBN: 978-3-642-19717-8.

[34] William J. Bowman and Amal Ahmed. “Noninterference for Free”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 101–113. ISBN: 9781450336697. DOI: 10.1145/2784731.2784733. URL: https://doi.org/10.1145/2784731.2784733.

[35] Elette Boyle, Niv Gilboa, and Yuval Ishai. “Function Secret Sharing”. In: *EUROCRYPT (2)*. Springer, 2015, pp. 337–367. DOI: 10.1007/978-3-662-46803-6_12. URL: https://www.iacr.org/archive/eurocrypt2015/90560300/90560300.pdf.

[36] Lennart Braun et al. “MOTION – A Framework for Mixed-Protocol Multi-Party Computation”. In: *ACM Trans. Priv. Secur.* 25.2 (Mar. 2022). ISSN: 2471-2566. DOI: 10.1145/3490390. URL: https://doi.org/10.1145/3490390.

[37] David Brumley and Dan Boneh. “Remote Timing Attacks Are Practical”. In: *USENIX Security*. 2003.

[38] Paul Bunn et al. “Efficient 3-Party Distributed ORAM”. In: *Security and Cryptography for Networks: 12th International Conference, SCN 2020, Amalfi, Italy, September 14–16, 2020, Proceedings*. Amalfi, Italy: Springer-Verlag, 2020, pp. 215–232. ISBN: 978-3-030-57989-0. DOI: 10.1007/978-3-030-57990-6_11. URL: https://doi.org/10.1007/978-3-030-57990-6_11.

[39] Niklas Büscher et al. “HyCC: Compilation of Hybrid Protocols for Practical Secure Computation”. In: *ACM CCS 2018: 25th Conference on Computer and Communications Security*. Ed. by David Lie et al. Toronto, ON, Canada: ACM Press, Oct. 2018, pp. 847–861. DOI: 10.1145/3243734.3243786.

[40] T-H. Hubert Chan et al. “Foundations of Differentially Oblivious Algorithms”. In: *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’19. San Diego, California: Society for Industrial and Applied Mathematics, 2019, pp. 2448–2467. URL: http://dl.acm.org/citation.cfm?id=3310435.3310585.

[41] Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. “On the Bayes risk in information-hiding protocols”. In: *Journal of Computer Security* 16.5 (2008).

[42] Benny Chor et al. “Private Information Retrieval”. In: *IEEE Symposium on Foundations of Computer Science (FOCS)*. 1995, pp. 41–50.

[43] Guillaume Claret et al. *Bayesian Inference for Probabilistic Programs via Symbolic Execution*. Tech. rep. MSR-TR-2012-86. Microsoft Research, 2012.

[44] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. "Quantifying information flow with beliefs". In: *Journal of Computer Security* 17.5 (2009), pp. 655–701.

[45] Michael R. Clarkson and Fred B. Schneider. "Quantification of Integrity". In: *2010 23rd IEEE Computer Security Foundations Symposium*. 2010, pp. 28–43. DOI: 10.1109/CSF.2010.10.

[46] Patrick Cousot and Radhia Cousot. "Static Determination of Dynamic Properties of Programs". In: *Proceedings of the Second International Symposium on Programming*. 1976.

[47] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages (POPL)*. 1977.

[48] Patrick Cousot and Nicolas Halbwachs. "Automatic Discovery of Linear Restraints Among Variables of a Program". In: *POPL*. 1978.

[49] Patrick Cousot and Michael Monerau. "Probabilistic Abstract Interpretation". In: *Proceedings of the European Symposium on Programming (ESOP)*. 2012.

[50] Luís Cruz-Filipe and Fabrizio Montesi. "A Core Model for Choreographic Programming". In: *Formal Aspects of Component Software*. Ed. by Olga Kouchnarenko and Ramtin Khosravi. Cham: Springer International Publishing, 2017, pp. 17–35. ISBN: 978-3-319-57666-4.

[51] Luís Cruz-Filipe and Fabrizio Montesi. "Procedural Choreographic Programming". In: *Formal Techniques for Distributed Objects, Components, and Systems*. Ed. by Ahmed Bouajjani and Alexandra Silva. Cham: Springer International Publishing, 2017, pp. 92–107. ISBN: 978-3-319-60225-7.

[52] Ivan Damgård et al. "Multiparty Computation from Somewhat Homomorphic Encryption". In: *CRYPTO*. 2012.

[53] Jesus A. De Loera et al. *LattE*. https://www.math.ucdavis.edu/~latte/. 2008.

[54] Daniel Demmler, Thomas Schneider, and Michael Zohner. "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation". In: *ISOC Network and Distributed System Security Symposium – NDSS 2015*. San Diego, CA, USA: The Internet Society, Feb. 2015.

[55] Dorothy E. Denning. "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5 (May 1976), pp. 236–243. ISSN: 0001-0782. DOI: 10.1145/360051.360056. URL: https://doi.org/10.1145/360051.360056.

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

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

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

[59] Shimon Even, Oded Goldreich, and Abraham Lempel. "A Randomized Protocol for Signing Contracts". In: *Commun. ACM* 28.6 (June 1985), pp. 637–647. ISSN: 0001-0782. DOI: 10.1145/3812.3818. URL: https://doi.org/10.1145/3812.3818.

[60] Brett Hemenway Falk and Rafail Ostrovsky. "Secure Merge with O(n log log n) Secure Operations". In: *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*. Ed. by Stefano Tessaro. Vol. 199. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 7:1–7:29. ISBN: 978-3-95977-197-9. DOI: 10.4230/LIPIcs.ITC.2021.7. URL: https://drops.dagstuhl.de/opus/volltexte/2021/14326.

[61] Matthias Felleisen and Robert Hieb. "The revised report on the syntactic theories of sequential control and state". In: *Theoretical computer science* 103.2 (1992), pp. 235–271.

[62] Christopher W. Fletcher et al. "Suppressing the Oblivious RAM timing channel while making information leakage and program efficiency trade-offs". In: *HPCA*. 2014.

[63] Martin Franz et al. "CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations". In: *Compiler Construction*. Ed. by Albert Cohen. Vol. 8409. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 244–249. DOI: 10.1007/978-3-642-54807-9_15. URL: http://dx.doi.org/10.1007/978-3-642-54807-9%5C_15.

[64] Marco Gaboardi et al. "Linear dependent types for differential privacy". In: *ACM SIGPLAN Notices*. Vol. 48. 1. ACM. 2013, pp. 357–370.

[65] Taher El Gamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". In: *IEEE Transactions on Information Theory* IT–31.4 (1985), pp. 469–472.

[66] Timon Gehr, Sasa Misailovic, and Martin Vechev. "PSI: Exact Symbolic Inference for Probabilistic Programs". In: *CAV*. 2016.

[67] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *41st Annual ACM Symposium on Theory of Computing*. Ed. by Michael Mitzenmacher. Bethesda, MD, USA: ACM Press, May 2009, pp. 169–178. DOI: 10.1145/1536414.1536440.

[68] Michèle Giry. "A categorical approach to probability theory". In: *Categorical Aspects of Topology and Analysis*. Ed. by B. Banaschewski. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 68–85. ISBN: 978-3-540-39041-1.

[69] J.A. Goguen and J. Meseguer. "Security policy and security models". In: *IEEE S & P*. 1982.

[70] Joseph A. Goguen and José Meseguer. "Security Policies and Security Models". In: *IEEE Symposium on Security and Privacy*. 1982, pp. 11–20.

[71] O. Goldreich. "Towards a Theory of Software Protection and Simulation by Oblivious RAMs". In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC '87. New York, New York, USA: Association for Computing Machinery, 1987, pp. 182–194. ISBN: 0897912217. DOI: 10.1145/28395.28416. URL: https://doi.org/10.1145/28395.28416.

[72] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority". In: *19th Annual ACM Symposium on Theory of Computing*. Ed. by Alfred Aho. New York City, NY, USA: ACM Press, May 1987, pp. 218–229. DOI: 10.1145/28395.28420.

[73] Oded Goldreich and Rafail Ostrovsky. "Software protection and simulation on oblivious RAMs". In: *J. ACM* (1996).

[74] S Goldwasser, S Micali, and C Rackoff. "The Knowledge Complexity of Interactive Proof-Systems". In: *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*. STOC '85. Providence, Rhode Island, USA: Association for Computing Machinery, 1985, pp. 291–304. ISBN: 0897911512. DOI: 10.1145/22145.22178. URL: https://doi.org/10.1145/22145.22178.

[75] Noah D. Goodman et al. "Church: a language for generative models". In: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*. 2008.

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

[77] Andrew D. Gordon et al. "Probabilistic Programming". In: *Conference on the Future of Software Engineering*. FOSE 2014. Hyderabad, India: ACM, 2014, pp. 167–181. ISBN: 978-1-4503-2865-4. DOI: 10.1145/2593882.2593900.

[78] Marco Guarnieri, Srdjan Marinovic, and David Basin. "Securing Databases from Probabilistic Inference". In: *Proc. IEEE Computer Security Foundations Symposium (CSF)*. 2017.

[79] Koki Hamada et al. "Practically Efficient Multi-Party Sorting Protocols from Comparison Sort Algorithms". In: *Proceedings of the 15th International Conference on Information Security and Cryptology*. ICISC'12. Seoul, Korea: Springer-Verlag, 2012, pp. 202–216. ISBN: 9783642376818. DOI: 10.1007/978-3-642-37682-5_15. URL: https://doi.org/10.1007/978-3-642-37682-5_15.

[80] Marcella Hastings et al. "SoK: General Purpose Compilers for Secure Multi-Party Computation". In: *S&P*. 2019.

[81] Nevin Heintze and Jon G. Riecke. "The SLam Calculus: Programming with Secrecy and Integrity". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '98. San Diego, California, USA: Association for Computing Machinery, 1998, pp. 365–377. ISBN: 0897919793. DOI: 10.1145/268946.268976. URL: https://doi.org/10.1145/268946.268976.

[82] Andrew K. Hirsch and Deepak Garg. *Pirouette: Higher-Order Typed Functional Choreographies*. 2021. arXiv: 2111.03484 [cs.PL].

[83] C. A. R. Hoare. "Communicating Sequential Processes". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585. URL: https://doi.org/10.1145/359576.359585.

[84] Matt Hoekstra. *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.

[85] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. "Resource Aware ML". In: *Proceedings of the 24th International Conference on Computer Aided Verification*. CAV'12. Berkeley, CA: Springer-Verlag, 2012, pp. 781–786. ISBN: 9783642314230. DOI: 10.1007/978-3-642-31424-7_64. URL: https://doi.org/10.1007/978-3-642-31424-7_64.

[86] Justin Hsu. "Probabilistic Couplings for Probabilistic Reasoning". In: *CoRR* abs/1710.09951 (2017). arXiv: 1710.09951. URL: http://arxiv.org/abs/1710.09951.

[87] Daniel Huang and Greg Morrisett. "An Application of Computable Distributions to the Semantics of Probabilistic Programming Languages". In: *Programming Languages and Systems*. Ed. by Peter Thiemann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 337–363. ISBN: 978-3-662-49498-1.

[88] Hans Hüttel et al. "Foundations of Session Types and Behavioural Contracts". In: *ACM Comput. Surv.* 49.1 (Apr. 2016). ISSN: 0360-0300. DOI: 10.1145/2873052. URL: https://doi.org/10.1145/2873052.

[89] R. Impagliazzo and S. Rudich. "Limits on the Provable Consequences of One-Way Permutations". In: *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*. STOC '89. Seattle, Washington, USA: Association for Computing Machinery, 1989, pp. 44–61. ISBN: 0897913078. DOI: 10.1145/73007.73012. URL: https://doi.org/10.1145/73007.73012.

[90] Yuval Ishai et al. "Extending Oblivious Transfers Efficiently". In: *Advances in Cryptology - CRYPTO 2003*. Ed. by Dan Boneh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 145–161. ISBN: 978-3-540-45146-4.

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

[92] Marcel Keller. "MP-SPDZ: A Versatile Framework for Multi-Party Computation". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1575–1590. ISBN: 9781450370899. URL: https://doi.org/10.1145/3372297.3417872.

[93] Florian Kerschbaum. "Automatically optimizing secure computation". In: *CCS*. 2011.

[94] Vladimir Klebanov. "Precise quantitative information flow analysis—a symbolic approach". In: *Theoretical Computer Science* 538 (2014), pp. 124–139.

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

[96] Paul Kocher et al. "Security As a New Dimension in Embedded System Design". In: *Proceedings of the 41st Annual Design Automation Conference*. DAC '04. Moderator-Ravi, Srivaths. 2004, pp. 753–760.

[97] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *Commun. ACM* 63.7 (June 2020), pp. 93–101. ISSN: 0001-0782. DOI: 10.1145/3399742. URL: https://doi.org/10.1145/3399742.

[98] Vladimir Kolesnikov and Thomas Schneider. "Improved Garbled Circuit: Free XOR Gates and Applications". In: *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*. ICALP '08. Reykjavik, Iceland: Springer-Verlag, 2008, pp. 486–498. ISBN: 9783540705826. DOI: 10.1007/978-3-540-70583-3_40. URL: https://doi.org/10.1007/978-3-540-70583-3_40.

[99] Boris Köpf and David Basin. "An Information-Theoretic Model for Adaptive Side-Channel Attacks". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2007. ISBN: 978-1-59593-703-2.

[100] Boris Köpf and Andrey Rybalchenko. "Approximation and Randomization for Quantitative Information-Flow Analysis". In: *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*. 2010.

[101] Boris Köpf and Andrey Rybalchenko. "Automation of Quantitative Information-Flow Analysis". In: *13th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM), 2013*. Vol. 7938. Lecture Notes in Computer Science. Springer, 2013, pp. 1–28.

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

[103] Martin Kučera et al. "Synthesis of Probabilistic Privacy Enforcement". In: *Proc. ACM Conference on Computer and Communications Security (CCS)*. 2017.

[104] Butler W. Lampson. "A Note on the Confinement Problem". In: *Commun. ACM* (1973).

[105] Sven Laur, Jan Willemson, and Bingsheng Zhang. "Round-Efficient Oblivious Database Manipulation". In: *ISC 2011: 14th International Conference on Information Security*. Ed. by Xuejia Lai, Jianying Zhou, and Hui Li. Vol. 7001. Lecture Notes in Computer Science. Xi'an, China: Springer, Heidelberg, Germany, Oct. 2011, pp. 262–277.

[106] Yehuda Lindell. "Secure Multiparty Computation". In: *Commun. ACM* 64.1 (Dec. 2020), pp. 86–96. ISSN: 0001-0782. DOI: 10.1145/3387108. URL: https://doi.org/10.1145/3387108.

[107] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *USENIX Security*. 2018.

[108] Chang Liu, Michael Hicks, and Elaine Shi. "Memory Trace Oblivious Program Execution". In: *Proceedings of the Computer Security Foundations Symposium (CSF)*. Winner of the 2014 NSA **Best Scientific Cybersecurity Paper** competition. June 2013. URL: http://www.cs.umd.edu/~mwh/papers/csf2013oram.pdf.

[109] Chang Liu et al. "Automating Efficient RAM-Model Secure Computation". In: *IEEE S & P*. May 2014.

[110] Chang Liu et al. "Automating Efficient RAM-Model Secure Computation". In: *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*. May 2014. URL: http://www.cs.umd.edu/~mwh/papers/ram-sc.pdf.

[111] Chang Liu et al. "GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation". In: *ASPLOS*. 2015.

[112] Chang Liu et al. "ObliVM: A Programming Framework for Secure Computation". In: *2015 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 2015, pp. 359–376. DOI: 10.1109/SP.2015.29.

[113] Isaac Liu et al. "A PRET microarchitecture implementation with repeatable timing and competitive performance". In: *ICCD*. 2012.

[114] Martin Maas et al. "Phantom: Practical Oblivious Computation in a Secure Processor". In: *CCS*. 2013.

[115] Piotr Mardziel et al. "Dynamic Enforcement of Knowledge-based Security Policies using Probabilistic Abstract Interpretation". In: *Journal of Computer Security* 21 (Oct. 2013), pp. 463–532.

[116] James L. Massey. "Guessing and Entropy". In: *Proc. IEEE Intl. Symposium on Information Theory (ISIT)*. 1994.

[117] Stephen McCamant and Michael D. Ernst. "Quantitative information flow as network flow capacity". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2008.

[118] Brian Milch et al. “Blog: Probabilistic models with unknown objects”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2005.

[119] Antoine Miné. “The Octagon Abstract Domain”. In: *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 2001.

[120] T. Minka et al. *Infer.NET 2.6*. Microsoft Research Cambridge. http://research.microsoft.com/infernet. 2014.

[121] David Molnar et al. “The Program Counter Security Model: Automatic Detection and Removal of Control-flow Side Channel Attacks”. In: *ICISC*. 2006.

[122] David Monniaux. “Abstract Interpretation of Probabilistic Semantics”. In: *Seventh International Static Analysis Symposium (SAS'00)*. Lecture Notes in Computer Science 1824. Springer Verlag, 2000, pp. 322–339. ISBN: 978-3-540-67668-3. DOI: 10.1007/978-3-540-45099-3_17.

[123] David Monniaux. “Analyse de programmes probabilistes par interprétation abstraite”. Thèse de doctorat. Université Paris IX Dauphine, 2001.

[124] Fabrizio Montesi. *Choreographic Programming*. English. 2013.

[125] Benjamin Mood et al. “Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation”. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 112–127.

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

[127] Chunyan Mu and David Clark. “An Interval-based Abstraction for Quantifying Information Flow”. In: *Elec. Notes in Theoretical Computer Science* 253.3 (2009), pp. 119–141.

[128] Andrew C. Myers. “JFlow: Practical Mostly-Static Information Flow Control”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, pp. 228–241. ISBN: 1581130953. DOI: 10.1145/292540.292561. URL: https://doi.org/10.1145/292540.292561.

[129] Praveen Narayanan et al. “Probabilistic Inference by Program Transformation in Hakaru (System Description)”. In: *Proc. Functional and Logic Programming*. 2016.

[130] Tri Minh Ngo, Mariëlle Stoelinga, and Marieke Huisman. “Effective verification of confidentiality for multi-threaded programs”. In: *Journal of computer security* 22.2 (2014).

[131] Jesper Buus Nielsen et al. “A New Approach to Practical Active-Secure Two-Party Computation”. In: *CRYPTO*. 2012.

[132] Olga Ohrimenko et al. "Oblivious Multi-party Machine Learning on Trusted Processors". In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC'16. Austin, TX, USA: USENIX Association, 2016, pp. 619–636. ISBN: 978-1-931971-32-4. URL: http://dl.acm.org/citation.cfm?id=3241094.3241143.

[133] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. "A Probabilistic Language Based on Sampling Functions". In: *ACM Trans. Program. Lang. Syst.* 31.1 (Dec. 2008), 4:1–4:46. ISSN: 0164-0925. DOI: 10.1145/1452044.1452048. URL: http://doi.acm.org/10.1145/1452044.1452048.

[134] James Parker, Niki Vazou, and Michael Hicks. "LWeb: Information Flow Security for Multi-Tier Web Applications". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290388. URL: https://doi.org/10.1145/3290388.

[135] Avi Pfeffer. *Figaro: An object-oriented probabilistic programming language*. Tech. rep. Charles River Analytics, 2000.

[136] Avi Pfeffer. "The Design and Implementation of IBAL: A General-Purpose Probabilistic Language". In: *Statistical Relational Learning*. Ed. by Lise Getoor and Benjamin Taskar. MIT Press, 2007.

[137] Mila Dalla Preda et al. "Dynamic Choreographies - Safe Runtime Updates of Distributed Applications". In: *COORDINATION*. 2015.

[138] Michael O. Rabin. *How To Exchange Secrets with Oblivious Transfer*. Harvard University Technical Report 81 talr@watson.ibm.com 12955 received 21 Jun 2005. 2005. URL: http://eprint.iacr.org/2005/187.

[139] Alexey Radul. "Report on the probabilistic language Scheme". In: *Proceedings of the Dynamic Languages Symposium (DLS)*. 2007.

[140] Norman Ramsey and Avi Pfeffer. "Stochastic Lambda Calculus and Monads of Probability Distributions". In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '02. Portland, Oregon: ACM, 2002, pp. 154–165. ISBN: 1-58113-450-9. DOI: 10.1145/503272.503288. URL: http://doi.acm.org/10.1145/503272.503288.

[141] Robert Rand and Steve Zdancewic. "VPHL". In: *Electron. Notes Theor. Comput. Sci.* 319.C (Dec. 2015), pp. 351–367. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2015.12.021. URL: http://dx.doi.org/10.1016/j.entcs.2015.12.021.

[142] Jaak Randmets. "Programming Languages for Secure Multi-party Computation Application Development". In: 2017.

[143] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. "Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations". In: *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*. May 2014. URL: http://www.cs.umd.edu/~mwh/papers/wysteria.pdf.

[144] Aseem Rastogi, Nikhil Swamy, and Michael Hicks. "Wys*: A DSL for Verified Secure Multi-party Computations". In: *Proceedings of the Symposium on Principles of Security and Trust (POST)*. Apr. 2019. URL: http://www.cs.umd.edu/~mwh/papers/wysstar.pdf.

[145] Aseem Rastogi et al. "Knowledge Inference for Optimizing Secure Multi-Party Computation". In: *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. PLAS '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 3–14. ISBN: 9781450321440. DOI: 10.1145/2465106.2465117. URL: https://doi.org/10.1145/2465106.2465117.

[146] Jason Reed and Benjamin C Pierce. "Distance makes the types grow stronger: a calculus for differential privacy". In: *ACM Sigplan Notices* 45.9 (2010), pp. 157–168.

[147] Ling Ren et al. "Design space exploration and optimization of path oblivious RAM in secure processors". In: *ISCA*. 2013.

[148] Alejandro Russo and Andrei Sabelfeld. "Securing interaction between threads and the scheduler". In: *CSF-W*. 2006.

[149] Alejandro Russo et al. "Closing Internal Timing Channels by Transformation". In: *Annual Asian Computing Science Conference (ASIAN)*. 2006.

[150] A. Sabelfeld and A. C. Myers. "Language-based Information-flow Security". In: *IEEE J.Sel. A. Commun.* 21.1 (Sept. 2006).

[151] Andrei Sabelfeld and David Sands. "Probabilistic noninterference for multi-threaded programs". In: *CSF-W*. 2000.

[152] Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes*. USA: Cambridge University Press, 2001. ISBN: 0521781779.

[153] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. "Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths". In: *Conference on Programming Language Design and Implementation*. PLDI. Seattle, Washington, USA, 2013. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462179.

[154] Tetsuya Sato et al. "Formal Verification of Higher-order Probabilistic Programs: Reasoning About Approximation, Convergence, Bayesian Inference, and Optimization". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 38:1–38:30. ISSN: 2475-1421. DOI: 10.1145/3290351. URL: http://doi.acm.org/10.1145/3290351.

[155] Berry Schoenmakers. *MPyC: Secure multiparty computation in Python*. Github. Feb. 2019. URL: https://github.com/lschoe/mpyc.

[156] Adam Ścibior, Zoubin Ghahramani, and Andrew D. Gordon. "Practical Probabilistic Programming with Monads". In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. Haskell '15. Vancouver, BC, Canada: ACM, 2015, pp. 165–176. ISBN: 978-1-4503-3808-0. DOI: 10.1145/2804302.2804317. URL: http://doi.acm.org/10.1145/2804302.2804317.

[157] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C". In: *ESEC/FSE*. 2005.

[158] Adi Shamir. "How to Share a Secret". In: *Commun. ACM* 22.11 (Nov. 1979), pp. 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: https://doi.org/10.1145/359168.359176.

[159] Claude Shannon. "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27 (1948).

[160] Elaine Shi et al. "Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost". In: *ASIACRYPT*. 2011.

[161] Calvin Smith, Justin Hsu, and Aws Albarghouthi. "Trace Abstraction Modulo Probability". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 39:1–39:31. ISSN: 2475-1421. DOI: 10.1145/3290352. URL: http://doi.acm.org/10.1145/3290352.

[162] Geoffrey Smith. "Probabilistic noninterference through weak probabilistic bisimulation". In: *CSF-W*. 2003.

[163] Geoffrey Smith. "On the Foundations of Quantitative Information Flow". In: *Proc. Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. 2009.

[164] Geoffrey Smith and Rafael Alpízar. "Secure Information Flow with Random Assignment and Encryption". In: *Workshop on Formal Methods in Security*. FMSE. 2006.

[165] Geoffrey Smith and Rafael Alpízar. "Fast Probabilistic Simulation, Nontermination, and Secure Information Flow". In: *PLAS*. 2007.

[166] Michael J. A. Smith. "Probabilistic Abstract Interpretation of Imperative Programs using Truncated Normal Distributions". In: *Elec. Notes in Theoretical Computer Science* (2008). ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.11.018.

[167] Ebrahim M. Songhori et al. "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits". In: *IEEE S & P*. 2015.

[168] Deian Stefan et al. *Flexible Dynamic Information Flow Control in the Presence of Exceptions*. 2012. DOI: 10.48550/ARXIV.1207.1457. URL: https://arxiv.org/abs/1207.1457.

[169] Emil Stefanov et al. "Path ORAM – an Extremely Simple Oblivious RAM Protocol". In: *CCS*. 2013.

[170] G. Edward Suh et al. "AEGIS: architecture for tamper-evident and tamper-resistant processing". In: *ICS*. 2003.

[171] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-order Execution". In: *USENIX Security*. 2018.

[172] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. "A Sound Type System for Secure Flow Analysis". In: *J. Comput. Secur.* 4.2-3 (Jan. 1996).

[173] Abraham Waksman. "A Permutation Network". In: *J. ACM* 15.1 (Jan. 1968), pp. 159–163. ISSN: 0004-5411. DOI: 10.1145/321439.321449. URL: https://doi.org/10.1145/321439.321449.

[174] Xiao Shaun Wang et al. "Oblivious Data Structures". In: *CCS*. 2014.

[175] Xiao Wang, Hubert Chan, and Elaine Shi. "Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 850–861. ISBN: 9781450338325. DOI: 10.1145/2810103.2813634. URL: https://doi.org/10.1145/2810103.2813634.

[176] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. *EMP-toolkit: Efficient MultiParty computation toolkit*. https://github.com/emp-toolkit. 2016.

[177] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. "Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 21–37. ISBN: 9781450349468. DOI: 10.1145/3133956.3134053. URL: https://doi.org/10.1145/3133956.3134053.

[178] Andrew Chi-Chih Yao. "Protocols for Secure Computations (Extended Abstract)". In: *FOCS*. 1982.

[179] Andrew Chi-Chih Yao. "How to Generate and Exchange Secrets (Extended Abstract)". In: *27th Annual Symposium on Foundations of Computer Science*. Toronto, Ontario, Canada: IEEE Computer Society Press, Oct. 1986, pp. 162–167. DOI: 10.1109/SFCS.1986.25.

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

[181] Samee Zahur and David Evans. *Obliv-C: A Language for Extensible Data-Oblivious Computation*. Cryptology ePrint Archive, Report 2018/706. https://eprint.iacr.org/2015/1153. 2015.

[182] Samee Zahur, Mike Rosulek, and David Evans. "Two Halves Make a Whole: Reducing Data Transfer in Garbled Circuits using Half Gates". In: *EUROCRYPT*. 2015.

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

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

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

[186] Danfeng Zhang et al. "A Hardware Design Language for Timing-Sensitive Information-Flow Security". In: *ASPLOS*. 2015.

[187] Hengchu Zhang et al. "Fuzzi: A Three-level Logic for Differential Privacy". In: *PACMPL* 3.ICFP (2019).

[188] Yihua Zhang, Aaron Steele, and Marina Blanton. "PICCO: a general-purpose compiler for private distributed computation". In: *ACM CCS 2013: 20th Conference on Computer and Communications Security*. Ed. by Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung. Berlin, Germany: ACM Press, Nov. 2013, pp. 813–826. DOI: 10.1145/2508859.2516752.

[189] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. "HIDE: an infrastructure for efficiently protecting information leakage on the address bus". In: *SIGARCH Comput. Archit. News* 32.5 (Oct. 2004).