Symphony: Expressive Secure Multiparty Computation with Coordination

Ian Sweet[a], David Darais[b], David Heath[c], William Harris[b], Ryan Estes[d], and Michael Hicks[a,e]

a University of Maryland

b Galois, Inc.

c Georgia Institute of Technology

d University of Vermont

e Amazon

Abstract Secure Multiparty Computation (MPC) refers to a family of cryptographic techniques with which mutually untrusting parties may compute arbitrary functions of their private inputs while revealing only the function output. MPC has broad application, but it can be hard to program MPCs correctly and efficiently, especially when they require coordinating disparate computational roles. We present Symphony, a new functional programming language for MPC whose programs can easily coordinate many parties, and whose *first-class shares* and *first-class party sets* provide unmatched language-level expressive power. Symphony generalizes the single-instruction, multiple-data (SIMD) semantics of prior MPC languages, and using a core language called λ-Symphony, we prove that this intuitive, generalized SIMD view of a program coincides with its actual distributed semantics. Thus the programmer can reason about her programs by reading them from top to bottom, even though in reality the program runs distributed across many machines. We implemented a prototype interpreter for Symphony and with it wrote a variety of MPC programs, finding that it can express optimized protocols that other languages cannot. We also measured the performance of the interpreter on several benchmark programs. Despite Symphony's added expressiveness, we find its performance is competitive with Obliv-C, a state-of-the-art two-party MPC framework for C.

ACM CCS 2012

- **Security and privacy → Logic and verification;**

Keywords secure multiparty computation, coordination, programming language, formal metatheory, distributed programming

In *The Art, Science, and Engineering of Programming*; 51 pages.

1 Introduction

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*. MPC technology today is hundreds of millions of times faster than technology from a mere 20 years ago, meaning that many more computations are now feasible. Modern frameworks are also increasingly convenient, offering programmers a familiar language in which to express their MPC programs [33, 2, 12, 35, 24, 30, 5].

Unfortunately, while these frameworks allow the programmer to express many MPCs, they fall short when confronted with problems that require non-trivial *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). This simplified view 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 such use-cases, the programmer must carefully coordinate the parties. 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 language Wysteria [27] does provide some coordination support, but lacks *expressiveness* and *ergonomics*. For example, individual parties may not delegate or relocate computations to other parties, and MPCs must be expressed in a rigid sublanguage that makes interleaving encrypted and plaintext computation awkward and inefficient.

Our Approach We present Symphony, an expressive MPC language that provides appropriate tools for coordinating MPCs with large numbers of parties. Symphony's most interesting language features are as follows:

- *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 set of parties that 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, in the style of choreographic programming languages [8, 9, 25, 23], Symphony retains a generalized SIMD character: We prove that the developer can reason about and debug her program as if it runs single threaded, even though in practice the program is deployed as multiple instances computing in parallel.

- *Party sets* are *first-class* values in the language: they can be put together and computed on at run-time, and used to annotate *par* blocks, thereby supporting highly expressive, dynamically determined coordination operations.
- *First-class shares* model multiparty-encrypted values in a way that allow the programmer to freely *delegate* computation and *reshare* encrypted values 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: *par* blocks and party sets are inspired by Wysteria, and first-class shares are inspired by the programming style of EMP [33] and Obliv-C [35]. Symphony is the first MPC language to carefully *combine* these features, and to *generalize* them, e.g., by allowing re-sharing and delegating already encrypted values to a different set of parties, and freely constructing and *computing* on party sets to refine coordinating actions. This combination of features is crucial for writing protocols in which parties may 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 2, Section 8).
- We present a core formalism for Symphony, called λ-Symphony—syntax (Section 3.1), *single-threaded* semantics (Section 3.2), and *distributed* semantics (Section 4). 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 5).
- We describe our prototype implementation for Symphony, which leverages EMP [33] and MOTION [5] as its cryptographic backends (Section 6).
- We discuss the 16 programs we have implemented in Symphony, showing that it provides *ergnomic* and *expressiveness* benefits compared to prior languages (Section 7.1). We present a complete implementation of the LWZ secure shuffling protocol [21], an important, highly optimized protocol that no prior MPC language can express (Section 2.2).
- We show that despite its expressiveness, Symphony enjoys good performance. On a set of kernel benchmarks running on a simulated LAN, Symphony takes a mean 1.15× the time taken by Obliv-C (Section 7.2).

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

2 Background and Overview

Most MPC frameworks express secure computation via an extension to a familiar programming language [33, 2, 12, 35, 24, 30]. 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 [35], in which two parties, A and B, wish to learn who 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 `recvs` 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* [34] to carry out these operations; other frameworks use *secret shares* [16, 4] or *homomorphic encryption* [14]. Regardless, most provide a similar SIMD-style programming model, either as library calls within an existing language (EMP [33], MPyC [30], MOTION [5]), or as a direct extension of that language (PICCO [36], ObliVM [22], and SCALE-MAMBA [2]). Sharemind [26] has developed its own C-like language, SecreC, which compiles to Sharemind assembly. Other works, such as CBMC-GC [12] and Frigate [24], compile subsets of C into circuits.

2.1 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 to localize execution, 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]

```
int a;
readInt(&a, ..., A);
int fa = ocCurrentParty() == A ? f(a) : 0;
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 $N \geq 2$ parties. In general, we might have dozens or 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 [27, 29], N-party languages like PICCO [36], Frigate [24], and SCALE-MAMBA [2] do not provide coordination mechanisms. Wysteria provides useful features but still suffers serious limitations, which we consider in detail in Section 7.

2.2 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 [21] showed how to implement a highly efficient secure shuffle among parties Q which is resilient to T corruptions, using a linear secret sharing scheme. The LWZ protocol implements a secure shuffle by repeatedly shuffling the elements among *committees*, which are strict subsets of the parties of

[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) it requires allocating memory on B for all of A's input and initializing the memory with dummy values.

Figure 1 A secure, N-party sorting procedure written in SYMPHONY. Uses the Shuffle-Then-Sort paradigm [17] 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.

```
1  party A B C D E
2
3  -- read input at p, secret-share to all in Q
4  def readShare Q p = par ({ p } \/ Q)
5    let i = par { p } read (array int) from "lwz.txt" in -- file local to each p
6    share [gmw, array int : { p } -> Q] i
7
8  def delegateShares P Q =
9    map (readShare Q) (psetToList P)
10
11 def shuffleWith Q S sharesQ = par (Q \/ S)
12   let sharesS = share [gmw, array int : Q -> S] sharesQ in
13   share [gmw, array int : S -> Q] (shuffle S sharesS)
14
15 def lwz Q sharesQ =
16   let t = 1 in
17   foldr (shuffleWith Q) sharesQ (subsets Q ((psetSize Q) - t))
18
19 def revealLte Q x y = reveal [gmw, bool : Q -> Q] x <= y
20
21 def secureSort Q sharesList =
22   let sharesQ  = par Q (arrayConcat sharesList) in
23   let shuffled = lwz Q sharesQ in
24   let sorted   = quickSort (revealLte Q) shuffled
25
26 def main () = par {A,B,C,D,E}
27   let Q = {A,B,C} in
28   let sharesList = delegateShares {A,B,C,D,E} Q in
29   let sorted     = secureSort Q sharesList in
30   ...
```

size $|Q| - T$. A committee is given shares of the input list and agrees on a random permutation, π; locally permutes its 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 some committee, which is sufficient to hide the global shuffle.[2] Figure 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,

[2] It is also necessary for security that $|Q| - T > T$ (equivalently, $|Q| \geq 2T + 1$) or else a corrupt committee of size $|Q| - T$ can collude to reveal the secret elements being shuffled.

only the listed parties execute the subsequent code .... Parties not in the given set skip the code 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, and can be computed on using the `case` expression. 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. The `subsets` function on line 17 computes on the party set `Q` using a `case` expression. 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`$[\phi, \tau : P \rightarrow Q]\ v$ 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`}∪ `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` ∈ `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`. Language support for resharing is unique to SYMPHONY, and it is critical for implementing LWZ. The `lwz` function securely shuffles `sharesQ`, which are shares among parties `Q`. The `foldr` on line 17 invokes `shuffleWith` on each subset `S` of `Q` (computed by `subsets`), which has size $|Q| - T$. 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 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 `quicksorts` 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 λ-Symphony: Syntax and Semantics

λ-Symphony is a minimal core language which captures the essential features of Symphony. This section presents its syntax and *single-threaded* (ST) semantics.

3.1 Syntax

The syntax of λ-Symphony is shown in Fig. 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. We isolate *atomic expressions* a as a sub-category 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 $x \; ? \; x \diamond x$ correspond to multiplexor expressions (written `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). 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 written $\lambda_z x.\; e$; the function body e may refer to itself via variable z. λ-Symphony also has mutable references, and primitives for I/O. λ-Symphony does not model lists or bundles because they are easily encoded; we explain how in the Appendix. 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 λ-Symphony, too, as discussed in Section 6; e.g., it permits sharing values of any type, and doing case analysis on encrypted sums.

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

Figure 2 λ-Symphony formal syntax.

$$
\begin{array}{rlll}
\ell \in \text{loc} & & & \textit{memory locations} \\
\psi \in \text{prot} & ::= & \cdot & \textit{cleartext} \\
& \mid & \texttt{enc\#}m & \textit{encrypted} \\
\gamma \in \text{env} & \triangleq & \text{var} \rightharpoonup \text{value} & \textit{value environment} \\
\delta \in \text{store} & \triangleq & \text{loc} \rightharpoonup \text{value} & \textit{value store} \\
u \in \text{loc-value} & ::= & i^{\psi} & \textit{integer/share value} \\
& \mid & p & \textit{party set value} \\
& \mid & \iota_i\ v & \textit{tagged union injection} \\
& \mid & \langle v, v \rangle & \textit{pairs} \\
& \mid & \langle \lambda_z x.\ e, \gamma \rangle & \textit{closures} \\
& \mid & \ell^{\#m} & \textit{reference} \\
v \in \text{value} & ::= & u@m & \textit{located value} \\
& \mid & \bigstar & \textit{opaque value}
\end{array}
$$

$$\boxed{\_\surd_m \in \text{loc-value} \to \text{loc-value}}$$

$$
\bigstar\surd_m \triangleq \bigstar \qquad\qquad (u@p)\surd_m \triangleq \begin{cases} u\surd_m@(p \cap m) & \textit{if } p \cap m \neq \varnothing \\ \bigstar & \textit{if } p \cap m = \varnothing \end{cases}
$$

Figure 3 λ-Symphony definitions and metafunctions used in formal semantics (partial).

3.2 Overview

The ST semantics for λ-Symphony 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 4, according to which parties may act independently. Thus, the ST semantics can serve as the basis of λ-Symphony 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, 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. Selected rules for both judgments are given in Figure 4; the full set of rules appears in the Appendix.

3.3 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, 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,[3] whereas $\texttt{enc}\#p$ represents an encrypted value shared among parties $B \in p$ (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.

3.4 Operational rules

Now we consider some of the operational rules.

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

$$\boxed{\gamma \vdash_m \delta, a \hookrightarrow \delta, v}$$

ST-Var
$$\frac{}{\gamma \vdash_m \delta, x \hookrightarrow \delta, \gamma(x)\surd_m}$$

ST-Lit
$$\frac{}{\begin{array}{l}\gamma \vdash_m \delta, i \hookrightarrow \delta, i@m \\ \gamma \vdash_m \delta, p \hookrightarrow \delta, p@m\end{array}}$$

ST-Int-Binop
$$\frac{\begin{array}{ll} i_1^{\psi}@m = \gamma(x_1)\surd_m & \\ i_2^{\psi}@m = \gamma(x_2)\surd_m & \vdash_m \psi \end{array}}{\gamma \vdash_m \delta, x_1 \odot x_2 \hookrightarrow \delta, [\![\odot]\!](i_1, i_2)^{\psi}@m}$$

ST-Ref
$$\frac{v = \gamma(x)\surd_m}{\gamma \vdash_m \delta, \texttt{ref}\ x \hookrightarrow \{\ell \mapsto v\} \uplus \delta, \ell^{\#m}@m}$$

ST-Deref
$$\frac{\ell^{\#q}@m = \gamma(x)\surd_m}{\gamma \vdash_m \delta, !x \hookrightarrow \delta, \delta(\ell)\surd_m}$$

ST-Assign
$$\frac{\begin{array}{l} \ell^{\#m}@m = \gamma(x_1)\surd_m \\ v = \gamma(x_2)\surd_m \end{array}}{\gamma \vdash_m \delta, x_1 := x_2 \hookrightarrow \delta[\ell \mapsto v], v}$$

ST-Share
$$\frac{\begin{array}{ll} p@m = \gamma(x_1)\surd_m & \vdash_p \psi \\ q@m = \gamma(x_2)\surd_m & q \neq \varnothing \\ i^{\psi}@p = \gamma(x_3)\surd_p & m = p \cup q \end{array}}{\gamma \vdash_m \delta, \texttt{share}[x_1 \to x_2]\ x_3 \hookrightarrow \delta, i^{\mathrm{enc}\#q}@q}$$

ST-Reveal
$$\frac{\begin{array}{ll} p@m = \gamma(x_1)\surd_m & \\ q@m = \gamma(x_2)\surd_m & q \neq \varnothing \\ i^{\mathrm{enc}\#p}@p = \gamma(x_3)\surd_p & m = p \cup q \end{array}}{\gamma \vdash_m \delta, \texttt{reveal}[x_1 \to x_2]\ x_3 \hookrightarrow \delta, i@q}$$

$$\boxed{\varsigma \longrightarrow \varsigma}$$

ST-Par
$$\frac{p@m = \gamma(x)\surd_m \qquad m \cap p \neq \varnothing}{m, \gamma, \delta, \kappa, \texttt{par}\ x\ e \longrightarrow m \cap p, \gamma, \delta, \kappa, e}$$

ST-ParEmpty
$$\frac{p@m = \gamma(x)\surd_m \qquad m \cap p = \varnothing \qquad \gamma' = \{x' \mapsto \bigstar\} \uplus \gamma}{m, \gamma, \delta, \kappa, \texttt{par}\ x\ e \longrightarrow m, \gamma', \delta, \kappa, x'}$$

Figure 4 λ-Symphony single-threaded semantics, selected rules.

Literals, Variables, Binding, Computation Rule ST-Var retrieves a variable's value from the environment and *(re)locates it* to the current mode m via $\gamma(x)\surd_m$. The function $\_\surd_m$ is given in Figure 3. For values $u@p$, $\_\surd_m$ 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; $u@p\surd_m$ also relocates the contents u to $u\surd_m$, 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. The full definition of relocation appears in the Appendix.

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 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 $i_1^\psi @m = \gamma(x_1)\surd_m$, 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.[4] 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. The distributed semantics also uses compatibility checks to ensure parties are in sync.

Par mode Operationally, $\texttt{par}\ x\ e$ evaluates e in mode $m \cap p$ where $p@m = \gamma(x)\surd_m$; 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.

$$
\begin{array}{l}
\texttt{par}\ \{A,B\}\ \texttt{let}\ x = \texttt{par}\ \{A\}\ 1\ \texttt{in} \\
\qquad\qquad\ \texttt{let}\ y = \texttt{par}\ \{B\}\ x\ \texttt{in} \\
\qquad\qquad\ \texttt{let}\ z = \texttt{par}\ \{C\}\ 2\ \texttt{in}\ x
\end{array}
$$

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 $\surd_{\{B\}}(1@\{A\})$ 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 $\surd_{\{A,B\}}(1@\{A\}) = 1@\{A\}$ per rule ST-Var.

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.

[4] Since cleartext inputs at different parties will not necessarily agree, the rules for handling I/O (not shown) require that the mode is a singleton party for the same reason.

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

```
par {A,B} let x = ref 0 in
          let _ = (par{A} x := 1) in
          let y = !x in ...
```

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 ... 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.

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; if x_3 is already an encrypted share (among p) then this operation will *reshare* it among q. The input parties p and share parties q must all be present in the mode m, and no other parties may be present (so $m = p \cup q$). Importantly, there is *no requirement* that $p \subseteq q$—this means that parties p may *delegate* the secure computation to parties q and not participate in it themselves. Both resharing and delegation are key strengths of Symphony not present in existing MPC languages. The share expressions in Figure 1 (critically) leverage both of these features.

Per rule ST-Reveal, $\texttt{reveal}[x_1 \to x_2]\ x_3$ reveals a shared encrypted value among parties p to a cleartext result at parties $q \neq \varnothing$, where x_1 evaluates to p, x_2 evaluates to q, and x_3 evaluates to the encrypted value. 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.

4 Distributed Semantics

This section presents λ-Symphony'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.

4.1 Configurations

A *distributed configuration* C collects the execution states of the individual parties in an MPC. As shown at the top of Figure 5, it consists of a finite map 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

$$\begin{array}{ll} \dot{v} \in \text{lval} ::= i^{\psi} \mid p \mid \ell^{\#m} \mid \iota_i\ \dot{v} & \dot{\varsigma} \in \text{lconfig} \ ::= m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, e \\ \qquad \mid\ \langle \dot{v}, \dot{v} \rangle \mid \langle \lambda_z x.\ e, \dot{\gamma} \rangle \mid \bigstar & C \in \text{dconfig} \triangleq \text{party} \rightharpoonup \text{lconfig} \end{array}$$

$$\boxed{\dot{\gamma} \vdash_m \dot{\delta}, a \hookrightarrow \dot{\delta}, \dot{v}}$$

$$\text{DS-Var}\ \frac{}{\dot{\gamma} \vdash_m \dot{\delta}, x \hookrightarrow \dot{\delta}, \dot{\gamma}(x)} \qquad \text{DS-Int-Binop}\ \frac{i_1^{\psi} = \dot{\gamma}(x_1) \qquad i_2^{\psi} = \dot{\gamma}(x_2) \qquad \vdash_m \psi}{\dot{\gamma} \vdash_m \dot{\delta}, x_1 \odot x_2 \hookrightarrow \dot{\delta}, [\![\odot]\!](i_1, i_2)^{\psi}}$$

$$\boxed{\dot{\varsigma} \longrightarrow_A \dot{\varsigma}}$$

$$\text{DS-Par}\ \frac{p = \dot{\gamma}(x) \qquad A \in p}{m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, \texttt{par}\ x\ e \longrightarrow_A m \cap p, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, e} \qquad \text{DS-ParEmpty}\ \frac{p = \dot{\gamma}(x) \qquad A \notin p \qquad \dot{\gamma}' = \{x' \mapsto \bigstar\} \uplus \dot{\gamma}}{m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, \texttt{par}\ x\ e \longrightarrow_A m, \dot{\gamma}', \dot{\delta}, \dot{\kappa}, x'}$$

$$\boxed{C \rightsquigarrow C}$$

$$\text{DS-Step}\ \frac{\dot{\varsigma} \longrightarrow_A \dot{\varsigma}'}{\{A \mapsto \dot{\varsigma}\} \uplus C \rightsquigarrow \{A \mapsto \dot{\varsigma}'\} \uplus C}$$

$$\text{DS-Share}\ \frac{\begin{array}{lll} \texttt{share}[x_1 \to x_2]\ x_3 = C(m).e & \vdash_p \psi & C' = \{A \mapsto (m, \{x \mapsto \dot{v}\} \uplus \dot{\gamma}, \dot{\delta}, \dot{\kappa}, x) \\ \qquad p = C(m).\dot{\gamma}(x_1) & m = C(m).m & \quad \mid C(A) = (m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, e), \\ \qquad q = C(m).\dot{\gamma}(x_2) & q \neq \varnothing & \quad A \in q \implies \dot{v} = i^{\text{enc}\#q}, \\ \qquad i^{\psi} = C(p).\dot{\gamma}(x_3) & m = p \cup q & \quad A \in p \wedge A \notin q \implies \dot{v} = \bigstar\} \end{array}}{C_0 \uplus C \rightsquigarrow C_0 \uplus C'}$$

$$\text{DS-Reveal}\ \frac{\begin{array}{lll} \texttt{reveal}[x_1 \to x_2]\ x_3 = C(m).e & & C' = \{A \mapsto (m, \{x \mapsto \dot{v}\} \uplus \dot{\gamma}, \dot{\delta}, \dot{\kappa}, x) \\ \qquad p = C(m).\dot{\gamma}(x_1) & m = C(m).m & \quad \mid C(A) = (m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, e), \\ \qquad q = C(m).\dot{\gamma}(x_2) & q \neq \varnothing & \quad A \in q \implies \dot{v} = i, \\ \qquad i^{\text{enc}\#p} = C(p).\dot{\gamma}(x_3) & m = p \cup q & \quad A \in p \wedge A \notin q \implies \dot{v} = \bigstar\} \end{array}}{C_0 \uplus C \rightsquigarrow C_0 \uplus C'}$$

Figure 5 λ-Symphony distributed semantics, selected rules. Note that judgment $\dot{\gamma} \vdash_m \dot{\delta}, a \hookrightarrow \dot{\delta}, \dot{v}$ is referred to by the elided rule DS-LetPop of the $\dot{\varsigma} \longrightarrow_A \dot{\varsigma}$ judgment, analogously to the single threaded semantics in Fig. 4.

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.

4.2 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 5. 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 $\gamma(x)\not\!/_m$, but rule DS-VAR's conclusion simply returns $\dot\gamma(x)$. Similarly, rule ST-INTBINOP's premise requires $i_1^\psi @m = \gamma(x_1)\not\!/_m$ 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 `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 `share[`$x_1 \to x_2$`]` x_3 and for DS-REVEAL it is `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.

[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.

$$\boxed{\_\lightning_A \in \text{loc-value} \rightarrow \text{lvalue} \,;\, \text{value} \rightarrow \text{lvalueenv} \rightarrow \text{lenv} \,;\, \text{store} \rightarrow \text{lstore} \,;\, \text{stack} \rightarrow \text{lstack}}$$

$$\begin{array}{cccc} i^{\psi}\lightning_A \triangleq i^{\psi} & \langle v_2, v_2\rangle\lightning_A \triangleq \langle v_1\lightning_A, v_2\lightning_B\rangle & u@p\lightning_A \triangleq \begin{cases} u\lightning_A & \textit{if} \quad A \in p \\ \bigstar & \textit{if} \quad A \notin p \end{cases} & \gamma\lightning_A(x) \triangleq \gamma(x)\lightning_A \\ p\lightning_A \triangleq p & \langle \lambda_z x.\ e, \gamma\rangle\lightning_A \triangleq \langle \lambda_z x.\ e, \gamma\lightning_A\rangle & & \delta\lightning_A(\ell) \triangleq \delta(\ell)\lightning_A \\ (\iota_i\ v)\lightning_A \triangleq \iota_i\ v\lightning_A & \ell^{\#m}\lightning_A \triangleq \ell^{\#m} & \bigstar\lightning_A \triangleq \bigstar & \end{array}$$

$$\top\lightning_A \triangleq \top \qquad (\langle \texttt{let}\ x = \square\ \texttt{in}\ e \mid m, \gamma\rangle :: \kappa)\lightning_A \triangleq \langle \texttt{let}\ x = \square\ \texttt{in}\ e \mid m, \gamma\lightning_A\rangle :: \kappa\lightning_A$$

$$(m, \gamma, \delta, \kappa, e)\lightning \triangleq \{A \mapsto m, \gamma\lightning_A, \delta\lightning_A, \kappa\lightning_A, e \mid A \in m\} \qquad \boxed{\_\lightning \in \text{config} \rightarrow \text{dconfig}}$$

Figure 6 Slicing metafunction; relates ST and DS semantics.

5 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 λ-Symphony program in terms of its ST semantics, even though in reality it will execute in a distributed fashion. Proofs are provided in Appendix B.2.

We relate a single-threaded configuration ς to a distributed one by *slicing* it, written $\varsigma\lightning$, which is defined in Figure 6. Each party A in the mode m of ς is mapped 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; $\_\lightning_A$ works much like $\_\checkmark_{\{A\}}$ but strips off all location annotations.

We can prove a full correspondence for programs whose execution trace concludes in normal form that is a *terminal state*. A terminal ST state has an empty stack and has reached a value; a DS state is terminal if all of its local configurations are terminal (see the Appendix. for a formal definition).

Theorem 5.1 (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 5.2 (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 5.2 does not rule out the possibility that ς gets stuck while $\varsigma\lightning$ never does. Consider this example

```
let x = par[A] <error> in par[B] <infinite loop>
```

In the ST semantics, this program gets stuck. In the DS semantics, A will only become *locally stuck* while B runs forever.[7] 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 5.3 (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, at which point an implementation could notify the other configurations of the problem.

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.

6 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 λ-Symphony, 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. Symphony supports semi-honest MPC over boolean circuits, supporting Yao's 2-party protocol with EMP toolkit [33] and the N-party GMW protocol with MOTION [5].

Neither EMP nor MOTION supports delegation or resharing, and MOTION does not support reactive MPC. The Symphony runtime, implemented in 2K lines of Rust, acts as a compatibility layer that enhances MPC backends with these features. The runtime adds support for delegation and resharing by adding semi-honest XOR sharing over Symphony parties. The XOR shares are converted to the native encrypted representaion of each backend as appropriate. The runtime adds support for reactive MPC to MOTION by adding a layer of caching for encrypted values.

The Symphony interpreter also generalizes λ-Symphony 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: λ-Symphony 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 multi-

[7] Examples like this are also the reason we cannot prove *full bisimulation*.

plexor. The precise value of `default` is determined when sharing, based on the type annotation τ on `share[`ϕ,τ:$P \rightarrow Q$`] default`.

A detailed description of all of the above can be found in Appendix A.1.

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. Appendix A.2 has more details about the standard library's contents.

7 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.

7.1 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 1. As points of comparison we consider if (or whether) versions of these programs could be implemented in Wysteria [27, 29] and/or Obliv-C [35], 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 express the programs in the **Features** column:

- *First-Class Party Sets* (`P`). Computation over party sets using `case`.
- *Reactive MPC* (`R`). Secure computation that depends on the decrypted result of a previous secure computation.
- *Synchronized Randomness* (`$`). Cryptographically secure random number generation, synchronized among a set of parties.
- *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.

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 2. `tournament`, `committee`, `lwz`, and `shuffle-qs` require secure computation over *First-Class Party Sets* which are computed dynamically, based on values only available at runtime.

Wysteria does not have support for *First-Class Party Sets*, *Delegation*, *Resharing*, or *Synchronized Randomness*. Wysteria does have support for party set values, but provides expression forms only for *creating* those values, not for *computing* with them. This precludes, for example, writing the `subsets` function in LWZ (Figure 1). We discuss

Table 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 first-class party sets, `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`). Appendix A.2 describes the programs in detail.

Program	#	Lang.	Features	L (C)	Description
`hamming` [12, 36]	2	`S,O,W?`		19 (629)	Find the Hamming distance of two strings.
`edit` [35]	2	`S,O,W?`		56 (1574)	Find the edit distance, by dynamic programming, of two strings.
`bio-match` [7]	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` [7]	2	`S,O,W?`		121 (3020)	Compute the mean and variance over the union and join of two databases.
`gcd` [19]	2	`S,O,W`		13 (416)	Compute the GCD of two numbers via Euclid's algorithm.
`richest` [27]	N	`S,W`		8 (176)	An N-party variant of the Millionaire's Problem.
`gps` [27]	N	`S,W`		40 (1213)	Compute the one-dimensional nearest neighbor for each of N parties.
`auction` [27]	N	`S,W`		34 (920)	Compute a second-price auction, revealing the second-highest bid to everyone and the highest bidder to auctioneer.
`median` [27]	2	`S,O?,W`	`R`	26 (566)	Compute the mixed-mode (reactive) median of a set of numbers.
`intersect` [27]	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` [31, 11]	N	`S,O`*	`$`	209 (5947)	Securely shuffle of an array, using N iterations of a Waksman permutation network.
`lwz` [21] (Section 2.2)	N	`S`	`P,$,S`	23 (745)	Securely shuffle of an array, using N reshares of a linear secret sharing scheme (LSSS).
`trivial-doram` [15, 6]	N	`S,O`*`,W?`		55 (1883)	A library for Oblivious RAM, adapted to MPC from trivial client-server ORAM.
`shuffle-qs` [17]	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.

this serious limitation further in Appendix A.3. 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: they all require *First-Class Party Sets* and `committee`, `lwz`, and `shuffle-qs` additionally require *Delegation* or *Resharing*.

Figure 7 The `gcd` program of Table 1 as written in SYMPHONY (top) and Wysteria (bottom).

```
def brec gcdr (a, b) =
  mux if (a == 0) then b
  else gcdr ((b % a), a)
def gcd = unroll gcdr (const 0) 93
```

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

Ergnomics SYMPHONY can also provide ergonomics benefits even when a program could be expressed in another language. We use Wysteria as a point of comparison, since it also has coordination features.

Like SYMPHONY, Wysteria coordinates parties by specifying which parties should execute in a given lexical scope: `let x =par(P)= e in` ... says that only parties in `P` should execute `e`. Wysteria prevents coordination errors statically, using a refinement type system. When variables are visible to more than one party, Wysteria's type system ensures that all parties agree on their contents.

Wysteria specifies MPCs explicitly: `let x =sec(P)= e in` ... says that the parties in `P` should jointly execute `e` as an MPC. When executed, the Wysteria interpreter translates `e` to a circuit and executes it with the GMW protocol, revealing the result to all parties. Rather than immediately reveal the final result, programmers can mark values as shares via `makesh`; these shares can be "reconstituted" via `combsh` in a later `sec` expression (among the same parties) to continue performing MPC over them. Once again, party annotations prevent accessing non-existent values. Moreover, the use of `sec` ensures all parties will synchronize at an MPC, avoiding deadlocks/hangs. Wysteria does not allow (recursive) function calls in `sec` scope, which makes secure algorithms requiring bounded recursion awkward and inefficient.

Consider the `gcd` function in Figure 7. This program computes the GCD of two encrypted integers among `{A,B}` in SYMPHONY (left) and Wysteria (right). The SYMPHONY implementation is idiomatic and efficient, thanks to flexible first-class shares. By contrast, Wysteria's implementation requires multiple `sec` blocks due to secure computation both before the call to `gcdr` (`b % a` on line 9) and after (`if (a == 0)` on line 13). The `gcd` function is recursive over encrypted values, so we must instead use bounded recursion according to a public upper bound (here, `93`) to ensure that it can be expressed as a circuit. SYMPHONY has a special `brec` keyword that facilitates bounded recursion: `def brec gcdr a b` expands into `def gcdr gcdr a b`, with the `gcdr` parameter shadowing the recursive binding of the same name. The call to `unroll gcdr (const 0) 93` takes `gcdr` and "unrolls" it by performing self-application 93 times, using `const 0` as the base case.

As mentioned above, Wysteria does not allow function calls within a `sec` scope. As a result, we are forced to enter and exit `sec` mode (lines 6,11) before and after 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 7-9 and 12-14) 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.

In sum, Symphony's support for first-class shares allows naturally recursive algorithms like GCD to be expressed idiomatically and efficiently under MPC. In contrast, Wysteria's requirement that shares *must* be computed within `sec` contexts, but that recursive functions *must not* be called in `sec` contexts makes these algorithms awkward and inefficient.

7.2 Performance

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 Appendix A.4, we ran on the first five programs in Table 1, which are well known and frequently referenced in the literature. We configured both Symphony and Obliv-C to use EMP's [33] 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, the sizes of garbled circuits generated by both frameworks are very similar (see Appendix A.4). Thus, we would expect a narrower gap with a more production-quality implementation.

Symphony's expressiveness also permits programmers to author algorithm-level optimization, 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 [**?** 28]. 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 resharing and comparisons/shuffling in the clear. As shown in Appendix A.5, the result is dramatically faster running times for all input sizes.

8 Related Work

The relationship of SYMPHONY to most prior work on MPC languages was discussed in Section 2, and to Wysteria in Section 7.1. One work we did not discuss is Viaduct [1], which compiles a Java-like language to secure distributed programs. SYMPHONY supports richer coordination patterns than Viaduct (e.g., LWZ in Figure 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 5), but has specific means to specify security policies, which may involve 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.

SYMPHONY's semantics of "generalized SIMD" bears resemblance to that of *choreography* languages [8, 9, 25, 23]. Choreographic programs are conceptually sequential, ensuring that `send` and `receive` operations are always matched up by combining them into a single expression. Pirouette [18] 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.

9 Conclusion

We have presented SYMPHONY, a new language for MPC with strong support for *coordination*. SYMPHONY's scoped *par* blocks, first-class party sets, and first-class shares—which support reactive computation, delegation, and resharing—provide unparalleled expressiveness and along with SYMPHONY's generalized SIMD semantics make coordination programming less error prone. We formalized core SYMPHONY and proved that the intuitive, single-threaded interpretation of a program coincides with its actual distributed semantics. Our prototype implementation exhibits performance competitive with existing systems, while uniquely enabling optimizations. We are working on new backends for SYMPHONY, a static type system, and support for oblivious data structures.

References

[1] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C. Myers, and Elaine Shi. 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, page 740–755, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454074.

[2] Abdelrahaman Aly, Marcel Keller, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. Scale-mamba. https://homes.esat.kuleuven.be/ nsmart/SCALE/, 2019.

[3] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999.

[4] 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*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press. doi:10.1145/62212.62213.

[5] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. Motion – a framework for mixed-protocol multi-party computation. *ACM Trans. Priv. Secur.*, 25(2), mar 2022. doi:10.1145/3490390.

[6] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-party distributed oram. In *Security and Cryptography for Networks: 12th International Conference, SCN 2020, Amalfi, Italy, September 14–16, 2020, Proceedings*, page 215–232, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-57990-6_11.

[7] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 847–861, Toronto, ON, Canada, October 15–19, 2018. ACM Press. doi:10.1145/3243734.3243786.

[8] Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. In Olga Kouchnarenko and Ramtin Khosravi, editors, *Formal Aspects of Component Software*, pages 17–35, Cham, 2017. Springer International Publishing.

[9] Luís Cruz-Filipe and Fabrizio Montesi. Procedural choreographic programming. In Ahmed Bouajjani and Alexandra Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 92–107, Cham, 2017. Springer International Publishing.

[10] 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, February 8–11, 2015. The Internet Society.

[11] Brett Hemenway Falk and Rafail Ostrovsky. Secure Merge with O(n log log n) Secure Operations. In Stefano Tessaro, editor, *2nd Conference on Information-*

Theoretic Cryptography (ITC 2021), volume 199 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:29, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2021/14326, doi:10.4230/LIPIcs.ITC.2021.7.

[12] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations. In Albert Cohen, editor, *Compiler Construction*, volume 8409 of *Lecture Notes in Computer Science*, pages 244–249. Springer Berlin Heidelberg, 2014. URL: http://dx.doi.org/10.1007/978-3-642-54807-9_15, doi:10.1007/978-3-642-54807-9_15.

[13] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Cbmc-gc: An ansi c compiler for secure two-party computations. In Albert Cohen, editor, *Compiler Construction*, pages 244–249, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[14] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178, Bethesda, MD, USA, May 31 – June 2, 2009. ACM Press. doi:10.1145/1536414.1536440.

[15] 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, page 182–194, New York, NY, USA, 1987. Association for Computing Machinery. doi:10.1145/28395.28416.

[16] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press. doi:10.1145/28395.28420.

[17] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *Proceedings of the 15th International Conference on Information Security and Cryptology*, ICISC'12, page 202–216, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-37682-5_15.

[18] Andrew K. Hirsch and Deepak Garg. Pirouette: Higher-order typed functional choreographies, 2021. arXiv:2111.03484.

[19] Muhammad Ishaq, Ana L. Milanova, and Vassilis Zikas. Efficient mpc via program analysis: A framework for efficient optimal mixing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1539–1556, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3319535.3339818.

[20] Florian Kerschbaum. Automatically optimizing secure computation. In *CCS*, 2011.

[21] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-efficient oblivious database manipulation. In Xuejia Lai, Jianying Zhou, and Hui Li, editors, *ISC 2011: 14th International Conference on Information Security*, volume 7001 of *Lecture*

Notes in Computer Science, pages 262–277, Xi'an, China, October 25–29, 2011. Springer, Heidelberg, Germany.

[22] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press. doi:10.1109/SP.2015.29.

[23] Fabrizio Montesi. Choreographic programming, 2013.

[24] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 112–127. IEEE, 2016.

[25] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies - safe runtime updates of distributed applications. In *COORDINATION*, 2015.

[26] Jaak Randmets. Programming languages for secure multi-party computation application development. 2017.

[27] 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.

[28] Aseem Rastogi, Piotr Mardziel, Michael Hicks, and Matthew A. Hammer. 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, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2465106.2465117.

[29] 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)*, April 2019. URL: http://www.cs.umd.edu/~mwh/papers/wysstar.pdf.

[30] Berry Schoenmakers. MPyC: secure multiparty computation in Python. Github, February 2019. URL: https://github.com/lschoe/mpyc.

[31] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, January 1968. doi:10.1145/321439.321449.

[32] 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, page 850–861, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2810103.2813634.

[33] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[34] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167,

Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press. doi:10.1109/SFCS.1986.25.

[35] Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2018/706, 2015. https://eprint.iacr.org/2015/1153.

[36] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: a general-purpose compiler for private distributed computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013: 20th Conference on Computer and Communications Security*, pages 813–826, Berlin, Germany, November 4–8, 2013. ACM Press. doi:10.1145/2508859.2516752.

A Implementation, Experiments, and Case Studies

A.1 Implementation

This section provides additional information about SYMPHONY's implementation (extending Section 6).

A.1.1 SYMPHONY Interpreter

This section provides additional information about the SYMPHONY Haskell interpreter. We highlight three interesting features: MPC over algebraic types, synchronized randomness, and the standard library.

MPC over Algebraic Types SYMPHONY generalizes λ-SYMPHONY'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 product (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` operation on encrypted sums is identical to `case`, and the `reveal` operation is works similarly to `share`.

We represent sums (variants) as tagged pairs. The λ-SYMPHONY 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 default 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.

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

To share a `default` value, $\texttt{share}[\phi, \tau : P \to Q]\ \texttt{default}$, we simply share instead the default value of the appropriate type τ. It is important for `mux-case` that the type τ be a monoid and that the default value is the identity. For example, when τ is `bool` the identity is `false` and the (monoidal) operation is `xor`. Likewise, when τ is `nat` the identity is 0 and the operation is +. The `reveal` operation on sum values works similarly.

The `mux-case` $\mathtt{sum}\langle b, v_0, v_1\rangle$ $\{\theta_1 \to e_1; \ldots; \theta_n \to e_n\}$ expression is the most interesting one we have to consider. The tag, b, is *encrypted* and so we cannot inspect it to determine if it matches a left or right injection pattern. So, what do we do? The `mux-case` proceeds by first filtering out any patterns $\theta_1, \ldots, \theta_n$ which are not either a left injection or a right injection pattern. Then, it *maps* over the remaining patterns $\theta_k \equiv \iota_i x$[8] by producing $\mathtt{mux}\ b\ \mathtt{then\ default\ else}\ v'_0$ when i is 0 and $\mathtt{mux}\ b\ \mathtt{then}\ v'_1\ \mathtt{else\ default}$ when i is 1 where v'_i is the result of evaluating e_k under the environment (γ) extended with $[x \mapsto v_i]$. At this point, we `add` together the list $v_1, \ldots, v_m$ (where $m \le n$ corresponds to the number of patterns in $\theta_1, \ldots, \theta_n$ which were either a left or right injection) of values just produced. The `add` procedure is the monoidal operation associated with the type, τ, of the values $v_1, \ldots, v_m$.

Consider the standard encoding of booleans as a sum of units: $\mathtt{bool} \coloneqq \mathtt{unit} + \mathtt{unit}$. We will take $\mathtt{sum}\langle\mathtt{true}, \bullet, \bullet\rangle$ 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: $\mathtt{mux}\ b\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \coloneqq \mathtt{mux\text{-}case\ sum}\langle b, \bullet, \bullet\rangle\ \{\iota_{\mathtt{true}}\ \bullet \to e_1;\ \iota_{\mathtt{false}}\ \bullet \to e_2\}$. Indeed, it does. In the following calculation, let v_i denote the result of evaluating e_i. Let τ be the type of v_1 and v_2, $\mathtt{id}_\tau$ denote the monoidal identity on τ and $\mathtt{add}_\tau$ denote the monoidal operation. Then, we have:

$$
\begin{aligned}
&\mathtt{mux\text{-}case\ sum}\langle\mathtt{true}, \bullet, \bullet\rangle\ \{\iota_0\ \bullet \to e_1; \iota_1\ \bullet \to e_2\}\\
&\equiv \mathtt{add}_\tau\ (\mathtt{mux\ true\ then}\ v_1\ \mathtt{else\ id}_\tau)\\
&\qquad\quad (\mathtt{mux\ true\ then\ id}_\tau\ \mathtt{else}\ v_2)\\
&\equiv \mathtt{add}_\tau\ v_1\ \mathtt{id}_\tau\\
&\equiv v_1
\end{aligned}
$$

MPC over algebraic types is an interesting problem, and there's a lot of unsolved problems in this space. We recommend that interested readers consult **?**] for a more complete treatment.

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 Symphony code below.

```
def randomSend P n =
  let rec sum' = fun Q ->
    case Q
    { {}           -> 0
    ; { p } \/ Q' ->
      send [nat : p -> P]
        (par { p } rand { p } nat) + (sum' Q')
    }
  in
  let sum = sum' P in
  sum % n
```

[8] We are ignoring nested patterns for simplicity. The procedure described here is straightforward to extend.

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.[9]

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. It establishes a shared seed among the parties `P` using the procedure above the first time they request a synchronized random number. Thereafter, the parties use a local, cryptographically secure pseudo-random generator which requires no communication.

The Standard Library 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, bounded recursion, and utilities for synchronized randomness. Next, we highlight some interesting, representative functions.

The coordination library primarily addresses common operations on principal sets, bundles, and their interaction. For example, the `bundleUpWith` function in Figure 8 takes a function `f` and a set of parties,`P`, and creates a bundle among the parties `P` by calling `f` locally on each party `p` in `P`. This function is useful for eliminating the boilerplate associated with the common pattern of reading an input from each party.

```
def readInputs P = bundleUpWith (fun _ -> read nat) P
```

The bounded recursion library in Symphony provides a critical function, `unroll`, which was used and briefly described in Figure 7. The full definition of `unroll` appears in Figure 8. As described in Section 7.1, the `unroll f init n` function is used in combination with the `brec` keyword to finitely "unroll" a function, `f`, `n` times before eventually bottoming out with a call to `init`. To illustrate, consider the `gcd` and `gcdr` functions from Section 7.1.

```
def brec gcdr (a, b) = mux if (a == 0) then b else gcdr ((b % a), a)
def gcd = unroll gcdr (const 0) 93
```

The signature `def brec` $f\ x_1 \ldots x_n$ desugars into the signature `def` $f\ f\ x_1 \ldots x_n$ so that any recursive call in the body of f now instead refers to the formal parameter f rather than the function being defined. By shadowing the recursive binding, we transparently turn any recursive function into one that can be used for *bounded* recursion. Here's what that looks like for `gcdr`:

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

Finally, we can compute the finite unrollings for 0 and 2 to see how `unroll` works in conjunction with functions desugared using `brec`.

[9] Readers may recognize that the random number generated by this procedure is slightly biased. When we use this procedure to establish a seed (described below) it is unbiased because the seed is a multiple of the word size of the machine (128 bits in practice).

■ **Figure 8** Selected functions from the standard library of Symphony.

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

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

def randNat P              = rand P nat
def randMaxNat P max       = randMax P nat max
def randRangeNat P min max = min + (randMaxNat P (max - min))

def shuffle P a =
  let n = size a in
  let shuffleRec = fun [shuffleRec] i ->
    if i < n - 1n then
      let j = randRangeNat P i n in
      let _ = swap a i j in
      shuffleRec (i + 1n)
    else ()
  in if n >= 2n then shuffleRec 0n else ()

def permutation P n =
  let a = upTo n in
  let _ = shuffle P a in
  a
```

```
def gcd = unroll gcdr (const 0) 0
        = if 0 == 0 then (const 0) else gcdr (unroll gcdr (const 0) (0-1))
        = const 0
```

```
def gcd = unroll gcdr (const 0) 2
        = if 2 == 0 then (const 0) else gcdr (unroll gcdr (const 0) (2-1))
        = gcdr (unroll gcdr (const 0) 1)
        = gcdr (if 1 == 0 then (const 0) else gcdr (unroll gcdr (const 0) (1-1)))
        = gcdr (gcdr (unroll gcdr (const 0) 0))
        = gcdr (gcdr (const 0))
```

A.1.2 Symphony Runtime

This section provides additional information about the Symphony Rust runtime. First, we briefly describe the common interface for MPC provided by the interpreter which is used to interact with the runtime. Then, we provide details about our approach to enhancing the supported MPC backends with delegation, resharing, and reactive MPC.

Encrypted values in Symphony are represented *abstractly*. The values encrypted with each Symphony protocol satisfy (the Haskell equivalent of) the following Java-ish interface.

```
1 public interface Enc <T> {
2   type Context;
3
4   // Turn the XOR share `share` into an abstract, encrypted `T`
5   void reflect(Context c, BaseValue share);
6
7   // Embed the cleartext constant `clear` as a `T`
8   void constant(Context c, BaseValue clear);
9
10  // Primitive Operations
11  T prim(Context c, Operation op, List<T> shares);
12
13  // Turn the abstract, encrypted `T` value, `enc`, into an XOR share
14  BaseValue reflect(Context c, T enc);
15 }
```

To add a new protocol to Symphony, one need only provide a new implementation of this interface. The `type Context` declaration on line 2 is not legal Java, and is meant to be evocative of an associated type in Rust or a type family in Haskell. Each protocol uniquely determines a type as its `Context`. The discussion on reactive MPC below gives more details. An advantage of this interface is that it allows us to ignore the differences between MPC based on secret sharing and MPC based on circuit garbling. For example, the MOTION backend implements MPC lazily by building up a circuit and executing it when a decryption is requested. In contrast, EMP implements MPC eagerly by garbling and executing gates as they are created. Both of these libraries, however, can be made to implement the interface above. Adding the enhancements necessary to do so is the subject of the next two sections.

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 uses the optimal conversion (`Y2B` [10]) 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` [10]) is not. Efficient conversion procedures for 2-party and N-party MPC protocols have been identified between Boolean sharing, Arithmetic sharing, and Boolean garbling [10, 5].

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).

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 Programs and Libraries

Here we describe some of the programs we have implemented in Symphony, tabulated in Table 1, in more detail. The language *features* required to implement each program are briefly discussed in Section 7.1.

The implementation of all the programs listed here can be found in Symphony's repository: https://github.com/plum-umd/symphony-lang/tree/master/res/Programming23. The templates used to generate Symphony programs for benchmarking can be found in https://github.com/plum-umd/symphony-lang/tree/master/experiments/templates.

hamming, **edit**, **bio-match**, **db-analytics**, and **gcd**, are 2-party programs, fully described in Table 1. We implemented these programs in Symphony and Obliv-C, which is limited to 2-party MPC. We believe they could be implemented in Wysteria.

richest, **gps**, **auction**, **median**, and **intersect** are also fully described in Table 1. They were previously implemented in Wysteria and we back-ported them to Symphony. **richest**, **gps**, and **auction** support N parties but otherwise require no special coordination. **median** and **intersect** support 2 parties and use *Reactive MPC* (referred to as "mixed-mode" by Wysteria) to improve their efficiency; we believe these could be implemented in Obliv-C.

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 *First-Class Party Sets* 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 *First-Class Party Sets* 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 is a protocol for securely shuffling an array of elements among N parties without revealing the permutation to any of them. The Waksman protocol implements a secure shuffle via repeated application of a classic Waksman *permutation network* [31]. 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. This protocol requires *Synchronized Randomness* to perform (non-secure) shuffles as a subroutine.

lwz also is a secure shuffle protocol; we presented it in Section 2.2 (Figure 1). In addition to *Synchronized Randomness* **lwz** requires *First-Class Party Sets* and *Resharing* to compute all the party sets of size $N - T$ and reshare the array of elements to and from these subsets. As far as we are are, no prior MPC language can directly express **lwz**.

trivial ORAM Trivial ORAM [32] allows secret shares to be used as addresses by simulating random access. The simulation performs a linear scan of the underlying array for every access (read or write). Multiplexors are used to choose which index to operate on. Trivial ORAM is a basic primitive which is used to construct more efficient ORAM implementations such as Circuit ORAM.

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 *First-Class Party Sets*, *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.

Figure 9 A simple Symphony function to elect a subset of $<= k$ parties from among `P`. A more sophisticated version of this function is used in the `committee` program (see Table 1).

```
def elect P k =
  if k == 0 then { }
  else case P
    { { }          -> { }
    ; { p } \/ P' -> { p } \/ (elect P' (k - 1))
    }
```

A.3 Committee Election in Wysteria

Wysteria does not support first-class party sets due to a lack of support for *computing* on them. We can see this in an attempt to port the `elect` function in Figure 9 from Symphony to Wysteria, which ultimately proves impossible. This limitation impacts Wysteria's ability to express coordination protocols compositionally.

The `elect P k` function "elects" a subset of at most `k` parties from the set `P`. A more featureful implementation might use a voting procedure or synchronized randomness to determine the subset, but this simple version will suffice for our purposes. Let's consider how we might express this function in Wysteria.

Attempt 1 The most natural way to express `elect` is through a signature that matches that of Symphony.

```
let elect (P : ps{true}) (k : nat) : ps{subeq P} = ??
```

Here we see a fairly typical OCaml-like signature with some type annotations. The type system of Wysteria is a refinement type system, but restricted so that refinements can only be placed on the party set type, `ps`. The set of refinements contains the standard propositional refinements `true` and conjunction (`and`), as well as special refinements `singl`, `subeq x`, and `eq x`. The `singl` refinement is satisfied by sets, P, that are singletons: $|P| = 1$. The `subeq Q` (`eq Q`) refinement is satisfied by party sets, P, that are a subset of (equal to) Q: $P \subseteq Q$.

In this signature for `elect`, we see that `P` has type `ps{true}` indicating that it is an unrefined party set. In addition, we see that `elect` should return a party set which is a subset of `P`. The fact that `P` is considered bound in the return type is a hallmark of dependent and refinement type systems.

Unfortunately, this attempt is dead on arrival. We cannot implement this signature because Wysteria simply has no expression form that corresponds to elimination (i.e. computation on) party sets. In our next attempt, we will try to circumvent this restriction by *encoding* party sets as arrays and computing over arrays instead.

Attempt 2 In this attempt, we will encode a party set `P` as an array containing singleton sets over elements belonging to `P` according to the type `array ps{singl and subeq P}`. This approach shows much more promise:

```
let elect (P  : ps{true})
          (Pe : array ps{singl and subeq P})
          (k  : nat)
          : array ps{singl and subeq P} =
  let ret = array [ k ] of (select Pe[0]) in
  let rec elect_loop i =
    if i < k then
      let _ = update ret[i] <- select Pe[i] in
      elect_loop (i + 1)
      else ()
  in
  let _ = elect_loop 0 in
  ret
```

This function loops through the first `k` elements of `Pe` and assigns them to a new array, `ret`, before returning `ret`. Contrary to our last attempt, the parameter `P` only exists in this signature to refine the type of `Pe` and the return type. All the computation is being performed over our encoding of party sets.

This version of `elect` is well-typed, and so we can try using it in a simple example.

```
let foo (Q : ps{true}) (Qe : array ps{singl and subeq Q}) : ... = ...

let P  = { !A, !B, !C } in
let Pe = array [ 3 ] of { !A } in
let _  = update Pe[1] <- { !B } in
let _  = update Pe[2] <- { !C } in
let Ce = elect P Pe 2 in
let ret =par(??)=
  foo ?? Ce
in ret
```

This little driver program declares `P` to contain the parties `A`, `B`, and `C` which are encoded in `Pe`. It uses the `elect` function above to produce an encoded committee, `Ce`, of 2 parties. At this point we would like just the committee members to run `foo`, but to do so we must fill the holes (`??`) with the actual party set represented by the encoding `Ce`. To do so, we define a `decode` function.

```
let decode (P : ps{true})
           (Pe : array ps{singl and subeq P})
           : ps{subeq P} =
  let n = length Pe in
  let rec decode_loop i acc =
    if i < n then
      let p = select Pe[i] in
      decode_loop (i + 1) (acc \/ p)
    else acc
  in decode_loop 0 { }
```

The `decode` function iterates over `Pe` and takes the union of all its elements. Since each of these elements is labeled as being a subset of `P`, Wysteria is able to deduce that the union of all the elements is also a subset of `P` as indicated by the return type `ps{subeq P}`.[10] Let's try using our `decode` function to fill in those holes from our earlier example.

```
let foo (Q : ps{true}) (Qe : array ps{singl and subeq Q}) : ... = ...

let P  = { !A, !B, !C } in
let Pe = array [ 3 ] of { !A } in
let _  = update Pe[1] <- { !B } in
let _  = update Pe[2] <- { !C } in
let Ce = elect P Pe 2 in
let C  = decode P Ce in
let ret =par(C)=
  foo C Ce
in ret
```

We can now enter an execution context containing only the parties `C` using `par(C)` mode. Unfortunately, the type system rejects this program at the call to `foo`. Why? Because Wysteria is unable to deduce that `Ce` has type `array ps{singl and subeq C}` as required by `foo`'s signature. Instead, the type of `Ce` is `array ps{singl and subeq P}`. We could imagine using a different signature for `foo`, but it is important for compositionality that `foo` take an encoded party set. Doing so enables `foo` to leverage any other functions that dynamically compute party sets according to our encoding. For example, `foo` might wish to call `elect` on its argument.

Of course, the critical issue is that the return type of `elect` is not precise enough, thereby preventing the encoded party set returned by `elect` from being passed to other functions that need to compute over party sets. At this point we are stuck. Wysteria does not offer any other facilities to circumvent the lack of an elimination form on party sets.

A.4 Performance 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. The results are summarized in Section 7.2; this section provides details of the experiments and results.

Experimental Setup For fair comparison, both Symphony and Obliv-C were configured to use EMP [33] 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 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 16 GB 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 were run on two simulated networks: a LAN (1 gbps bandwidth, <1 ms RTT latency) and a

[10] Wysteria does not support the `length` primitive on arrays that we used to compute `n`. This could be easily added, and allows us to avoid cluttering up our signatures with array lengths.

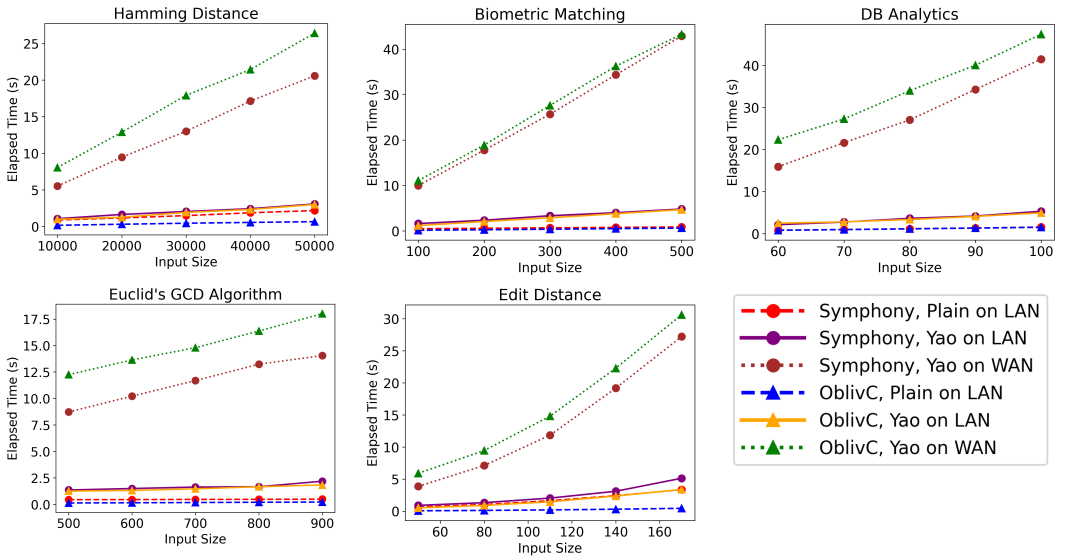

Figure 10 End-to-end execution time of 5 programs, averaged over five samples (lower is better). **LAN** is a simulated 1 gbps connection with no delay. **WAN** is a simulated 100 mbps connection with a 50 ms RTT latency. **Yao** and **Plain** protocols use EMP's `sh2pc` (semi-honest, two-party) and `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.

WAN (100Mbps bandwidth, 50 ms 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 10 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 λ-SYMPHONY, 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 $> 5\,ms$. 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 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.

A.5 Security vs. performance for secure-shuffle

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.

Table 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

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

SYMPHONY Execution Time Figure 11 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

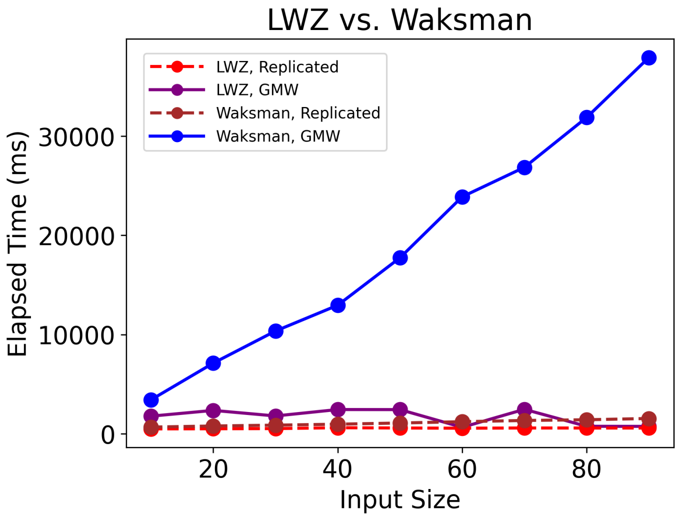

Figure 11 End-to-end execution time of Waksman vs LWZ shuffle over three parties, averaged over five samples (lower is better). The **Replicated** protocol executes the program without cryptography, executing operations in the clear. The **GMW** protocol uses the Symphony implementation of GMW which uses MOTION as a backend. **Input Size** indicates the length of the integer list provided as input by each party.

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 > 4 GB 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.

$$\kappa \in \text{stack} ::= \top \mid \langle \texttt{let}\ x = \square\ \texttt{in}\ e \mid m, \gamma \rangle :: \kappa \qquad\qquad \varsigma \in \text{config} ::= m, \gamma, \delta, \kappa, e$$

$$\boxed{\gamma \vdash_m \delta, a \hookrightarrow \delta, v}$$

$$\textsc{ST-Var}\ \frac{}{\gamma \vdash_m \delta, x \hookrightarrow \delta, \gamma(x)\angle_m}$$

$$\textsc{ST-Lit}\ \frac{}{\begin{array}{c}\gamma \vdash_m \delta, i \hookrightarrow \delta, i@m \\ \gamma \vdash_m \delta, p \hookrightarrow \delta, p@m\end{array}}$$

$$\textsc{ST-Int-Binop}\ \frac{\begin{array}{c} i_1^{\psi}@m = \gamma(x_1)\angle_m \\ i_2^{\psi}@m = \gamma(x_2)\angle_m \qquad \vdash_m \psi \end{array}}{\gamma \vdash_m \delta, x_1 \odot x_2 \hookrightarrow \delta, [\![\odot]\!](i_1, i_2)^{\psi}@m}$$

$$\textsc{ST-PSet-Binop}\ \frac{\begin{array}{c} p_1@m = \gamma(x_1)\angle_m \\ p_2@m = \gamma(x_2)\angle_m \end{array}}{\gamma \vdash_m \delta, x_1 \cup x_2 \hookrightarrow \delta, (p_1 \cup p_2)@m}$$

$$\textsc{ST-Mux}\ \frac{\begin{array}{c} i_1^{\psi}@m = \gamma(x_1)\angle_m \\ i_2^{\psi}@m = \gamma(x_2)\angle_m \\ i_3^{\psi}@m = \gamma(x_3)\angle_m \qquad \vdash_m \psi \end{array}}{\gamma \vdash_m \delta, x_1\ ?\ x_2 \diamond x_3 \hookrightarrow \delta, \text{cond}(i_1, i_2, i_3)^{\psi}@m}$$

$$\textsc{ST-Pair}\ \frac{\begin{array}{c} v_1 = \gamma(x_1)\angle_m \\ v_2 = \gamma(x_2)\angle_m \end{array}}{\gamma \vdash_m \delta, \langle x_1, x_2 \rangle \hookrightarrow \delta, \langle v_1, v_2 \rangle@m}$$

$$\textsc{ST-Proj}\ \frac{\langle v_1, v_2 \rangle@m = \gamma(x)\angle_m}{\gamma \vdash_m \delta, \pi_i\ x \hookrightarrow \delta, v_i}$$

$$\textsc{ST-Inj}\ \frac{v = \gamma(x)\angle_m}{\gamma \vdash_m \delta, \iota_i\ x \hookrightarrow \delta, (\iota_i\ v)@m}$$

$$\textsc{ST-Fun}\ \frac{}{\gamma \vdash_m \delta, \lambda_z x.\ e \hookrightarrow \delta, \langle \lambda_z x.\ e, \gamma \rangle@m}$$

$$\textsc{ST-Ref}\ \frac{v = \gamma(x)\angle_m}{\gamma \vdash_m \delta, \texttt{ref}\ x \hookrightarrow \{\ell \mapsto v\} \uplus \delta, \ell^{\#m}@m}$$

$$\textsc{ST-Deref}\ \frac{\ell^{\#q}@m = \gamma(x)\angle_m}{\gamma \vdash_m \delta, !x \hookrightarrow \delta, \delta(\ell)\angle_m}$$

$$\textsc{ST-Assign}\ \frac{\begin{array}{c} \ell^{\#m}@m = \gamma(x_1)\angle_m \\ v = \gamma(x_2)\angle_m \end{array}}{\gamma \vdash_m \delta, x_1 := x_2 \hookrightarrow \delta[\ell \mapsto v], v}$$

$$\textsc{ST-Read}\ \frac{|m| = 1}{\gamma \vdash_m \delta, \texttt{read} \hookrightarrow \delta, i@m}$$

$$\textsc{ST-Write}\ \frac{i@m = \gamma(x)\angle_m \qquad |m| = 1}{\gamma \vdash_m \delta, \texttt{write}\ x \hookrightarrow \delta, 0@m}$$

$$\textsc{ST-Share}\ \frac{\begin{array}{cc} p@m = \gamma(x_1)\angle_m & \vdash_p \psi \\ q@m = \gamma(x_2)\angle_m & q \neq \varnothing \\ i^{\psi}@p = \gamma(x_3)\angle_p & m = p \cup q \end{array}}{\gamma \vdash_m \delta, \texttt{share}[x_1 \rightarrow x_2]\ x_3 \hookrightarrow \delta, i^{\text{enc}\#q}@q}$$

$$\textsc{ST-Reveal}\ \frac{\begin{array}{cc} p@m = \gamma(x_1)\angle_m & \\ q@m = \gamma(x_2)\angle_m & q \neq \varnothing \\ i^{\text{enc}\#p}@p = \gamma(x_3)\angle_p & m = p \cup q \end{array}}{\gamma \vdash_m \delta, \texttt{reveal}[x_1 \rightarrow x_2]\ x_3 \hookrightarrow \delta, i@q}$$

$$\boxed{\varsigma \longrightarrow \varsigma}$$

$$\textsc{ST-Case-Inj}\ \frac{(\iota_i\ v)@m = \gamma(x_1)\angle_m}{m, \gamma, \delta, \kappa, \texttt{case}\ x_1\ \{x_2.e_1\}\{x_2.e_2\} \longrightarrow m, \{x_2 \mapsto v\} \uplus \gamma, \delta, \kappa, e_i}$$

$$\textsc{ST-Case-PSet-Emp}\ \frac{\varnothing@m = \gamma(x_1)\angle_m}{m, \gamma, \delta, \kappa, \texttt{case}\ x_1\ \{.e_1\}\{x_2 x_3.e_2\} \longrightarrow m, \gamma, \delta, \kappa, e_1}$$

$$\textsc{ST-Case-PSet-Cons}\ \frac{(\{A\} \uplus p)@m = \gamma(x_1)\angle_m}{m, \gamma, \delta, \kappa, \texttt{case}\ x_1\ \{.e_1\}\{x_2 x_3.e_2\} \longrightarrow m, \{x_2 \mapsto \{A\}, x_3 \mapsto p\} \uplus \gamma, \delta, \kappa, e_2}$$

$$\textsc{ST-Par}\ \frac{p@m = \gamma(x)\angle_m \qquad m \cap p \neq \varnothing}{m, \gamma, \delta, \kappa, \texttt{par}\ x\ e \longrightarrow m \cap p, \gamma, \delta, \kappa, e}$$

$$\textsc{ST-ParEmpty}\ \frac{p@m = \gamma(x)\angle_m \qquad m \cap p = \varnothing \qquad \gamma' = \{x' \mapsto \bigstar\} \uplus \gamma}{m, \gamma, \delta, \kappa, \texttt{par}\ x\ e \longrightarrow m, \gamma', \delta, \kappa, x'}$$

$$\textsc{ST-App}\ \frac{v_1 = \gamma(x_1)\angle_m \qquad v_2 = \gamma(x_2)\angle_m \qquad \langle \lambda_z x.\ e, \gamma' \rangle@m = v_1}{m, \gamma, \delta, \kappa, x_1\ x_2 \longrightarrow m, \{z \mapsto v_1, x \mapsto v_2\} \uplus \gamma', \delta, \kappa, e}$$

$$\textsc{ST-LetPush}\ \frac{\kappa' = \langle \texttt{let}\ x = \square\ \texttt{in}\ e_2 \mid m, \gamma \rangle :: \kappa}{m, \gamma, \delta, \kappa, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow m, \gamma, \delta, \kappa', e_1}$$

$$\textsc{ST-LetPop}\ \frac{\gamma \vdash_m \delta, a \hookrightarrow \delta', v \qquad \kappa = \langle \texttt{let}\ x = \square\ \texttt{in}\ e \mid m', \gamma' \rangle :: \kappa'}{m, \gamma, \delta, \kappa, a \longrightarrow m', \{x \mapsto v\} \uplus \gamma', \delta', \kappa', e}$$

Figure 12 λ-Symphony single-threaded semantics. Premises highlighted are required only for lazy MPC evaluation.

B Metatheory

B.1 Single-threaded Semantics

Figure 12 shows the complete single-threaded semantics which are discussed in Section 3.2.

Notice that rules for handling I/O 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).

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.

Symphony directly supports lists and arrays; in λ-Symphony 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 $@\{A, B\}$.)

B.2 Proof Sketches for Correspondence Theorems

To prove theorems Theorem 5.1, Theorem 5.2 and Theorem 5.3 given in Section 5, we first formalize key definitions.

Definition B.1 (Terminal State).

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

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*.

Definition B.2 (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 B.3 (Locally stuck).

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

Now we establish a number of key lemmas. Our proof approach for Theorem 5.1 largely follows the proof approach from Wysteria [27], and our proof approach for Theorem 5.2 and Theorem 5.3—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 B.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 B.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 B.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 5.1 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 B.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 B.3 and Lemma B.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 B.3 we know $\varsigma\lightning \rightsquigarrow^* \varsigma'\lightning$, and by Lemma B.2 we know $C = \varsigma'\lightning$. □

Our proof of Theorem 5.2 also follows from the lemmas and theorem proven thus far:

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

1. By Theorem 5.1 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 B.3 we know that there exists a distributed trace that also diverges. By Lemma B.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 B.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 5.1, 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 B.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 5.3 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 B.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 B.4 with $\varsigma\lightning$ as the common ancestor we have C' locally stuck. □

Theorem 5.1 captures the same metatheoretical properties proved of prior work (Wysteria [27]), whereas Theorems 5.2 and 5.3 are refinements of divergence-soundness and stuck-state-soundness results novel to our work.

B.3 Detailed Proofs for Key Lemmas

In this section, we prove the key meta-theoretic properties of the distributed semantics, namely forward simulation (Appendix B.3.1) and confluence (Appendix B.3.2), along with their corollaries. The full DS-semantics rules are given in Figure 13.

B.3.1 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 B.5 (Forward Simulation-Step). *If $\varsigma \rightarrow \varsigma'$, then $\varsigma \lightning \rightsquigarrow^* \varsigma' \lightning$.*

Proof. $\varsigma \rightarrow \varsigma'$, 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 $\varsigma \rightarrow \varsigma'$:

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 $x_1 \oplus x_2$, $\ddot{i}_1@m = \gamma(x_1) \swarrow_m$, and $\ddot{i}_2@m = \gamma(x_1) \swarrow_m$ 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 $\texttt{mux if } x_1 \texttt{ then } x_2 \texttt{ else } x_3$, $\ddot{i}_1@m = \gamma(x_1) \swarrow_m$, $\ddot{i}_2@m = \gamma(x_2) \swarrow_m$, and $\ddot{i}_3@m = \gamma(x_3) \swarrow_m$ 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 $x_1 \oplus x_2$, $i_1^{\mathsf{enc}\#m}@m = \gamma(x_1) \swarrow_m$, and $i_2^{\mathsf{enc}\#m}@m = \gamma(x_1) \swarrow_m$ (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 $\varsigma\lightning|_{\mathsf{parties}\setminus m}$ 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 $\texttt{mux if } x_1 \texttt{ then } x_2 \texttt{ else } x_3$, $i_1^{\mathsf{enc}\#m}@m = \gamma(x_1) \swarrow_m$, $i_2^{\mathsf{enc}\#m}@m = \gamma(x_2) \swarrow_m$, and $i_3^{\mathsf{enc}\#m}@m = \gamma(x_3) \swarrow_m$ 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 B.5.

Lemma B.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 B.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.

□

B.3.2 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 B.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 B.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 B.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 B.8) is a direct consequence of fundamental properties of general transition and rewrite systems.

Lemma B.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 [3] 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 B.8.1 (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 B.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. □

$$
\begin{array}{lll}
\dot{v} \in \text{lval} ::= i^{\psi} \mid p \mid \ell^{\#m} & \dot{\gamma} \in \text{lenv} \triangleq \text{var} \rightharpoonup \text{lval} & \dot{\varsigma} \in \text{lconfig} ::= m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, e \\
\quad \mid \iota_i \ \dot{v} \mid \langle \dot{v}, \dot{v} \rangle & \dot{\delta} \in \text{lstore} \triangleq \text{loc} \rightharpoonup \text{lval} & C \in \text{dconfig} \triangleq \text{party} \rightharpoonup \text{lconfig} \\
\quad \mid \langle \lambda_z x.\ e, \dot{\gamma} \rangle \mid \bigstar & \dot{\kappa} \in \text{lstack} ::= \top \mid \langle \texttt{let}\ x = \square\ \texttt{in}\ e \mid m, \dot{\gamma} \rangle :: \dot{\kappa} &
\end{array}
$$

$$\boxed{\dot{\gamma} \vdash_m \dot{\delta}, a \hookrightarrow \dot{\delta}, \dot{v}}$$

DS-Var
$$\frac{}{\dot{\gamma} \vdash_m \dot{\delta}, x \hookrightarrow \dot{\delta}, \dot{\gamma}(x)}$$

DS-Lit
$$\frac{}{\begin{array}{c}\dot{\gamma} \vdash_m \dot{\delta}, i \hookrightarrow \dot{\delta}, i \\ \dot{\gamma} \vdash_m \dot{\delta}, p \hookrightarrow \dot{\delta}, p\end{array}}$$

DS-Int-Binop
$$\frac{i_1^{\psi} = \dot{\gamma}(x_1) \qquad i_2^{\psi} = \dot{\gamma}(x_2) \qquad \vdash_m \psi}{\dot{\gamma} \vdash_m \dot{\delta}, x_1 \odot x_2 \hookrightarrow \dot{\delta}, [\![\odot]\!](i_1, i_2)^{\psi}}$$

DS-PSet-Binop
$$\frac{p_1 = \dot{\gamma}(x_1) \qquad p_2 = \dot{\gamma}(x_2)}{\dot{\gamma} \vdash_m \dot{\delta}, x_1 \cup x_2 \hookrightarrow \dot{\delta}, p_1 \cup p_2}$$

DS-Mux
$$\frac{i_1^{\psi} = \dot{\gamma}(x_1) \qquad i_2^{\psi} = \dot{\gamma}(x_2) \qquad i_3^{\psi} = \dot{\gamma}(x_3) \qquad \vdash_m \psi}{\dot{\gamma} \vdash_m \dot{\delta}, x_1 \mathrel{?} x_2 \diamond x_3 \hookrightarrow \dot{\delta}, \text{cond}(i_1, i_2, i_3)^{\psi}}$$

DS-Pair
$$\frac{\dot{v}_1 = \dot{\gamma}(x_1) \qquad \dot{v}_2 = \dot{\gamma}(x_2)}{\dot{\gamma} \vdash_m \dot{\delta}, \langle x_1, x_2 \rangle \hookrightarrow \dot{\delta}, \langle \dot{v}_1, \dot{v}_2 \rangle}$$

DS-Proj
$$\frac{\langle \dot{v}_1, \dot{v}_2 \rangle = \dot{\gamma}(x)}{\dot{\gamma} \vdash_m \dot{\delta}, \pi_i\ x \hookrightarrow \dot{\delta}, \dot{v}_i}$$

DS-Inj
$$\frac{\dot{v} = \dot{\gamma}(x)}{\dot{\gamma} \vdash_m \dot{\delta}, \iota_i\ x \hookrightarrow \dot{\delta}, (\iota_i\ \dot{v})}$$

DS-Fun
$$\frac{}{\dot{\gamma} \vdash_m \dot{\delta}, \lambda_z x.\ e \hookrightarrow \dot{\delta}, \langle \lambda_z x.\ e, \dot{\gamma} \rangle}$$

DS-Ref
$$\frac{\dot{v} = \dot{\gamma}(x)}{\dot{\gamma} \vdash_m \dot{\delta}, \texttt{ref}\ x \hookrightarrow \{\ell \mapsto \dot{v}\} \uplus \dot{\delta}, \ell^{\#m}}$$

DS-Deref
$$\frac{\ell^{\#q} = \dot{\gamma}(x)}{\dot{\gamma} \vdash_m \dot{\delta}, !x \hookrightarrow \dot{\delta}, \dot{\delta}(\ell)}$$

DS-Assign
$$\frac{\ell^{\#m} = \dot{\gamma}(x_1) \qquad \dot{v} = \dot{\gamma}(x_2)}{\dot{\gamma} \vdash_m \dot{\delta}, x_1 := x_2 \hookrightarrow \dot{\delta}[\ell \mapsto \dot{v}], \dot{v}}$$

DS-Read
$$\frac{|m| = 1}{\dot{\gamma} \vdash_m \dot{\delta}, \texttt{read} \hookrightarrow \dot{\delta}, i}$$

DS-Write
$$\frac{i = \dot{\gamma}(x) \qquad |m| = 1}{\dot{\gamma} \vdash_m \dot{\delta}, \texttt{write}\ x \hookrightarrow \dot{\delta}, 0}$$

$$\boxed{\dot{\varsigma} \longrightarrow_A \dot{\varsigma}}$$

DS-Case-Inj
$$\frac{(\iota_i\ \dot{v}) = \dot{\gamma}(x_1)}{m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, \texttt{case}\ x_1\ \{x_2.e_1\}\{x_2.e_2\} \longrightarrow_A m, \{x_2 \mapsto \dot{v}\} \uplus \dot{\gamma}, \dot{\delta}, \dot{\kappa}, e_i}$$

DS-Case-PSet-Emp
$$\frac{\varnothing = \dot{\gamma}(x_1)}{m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, \texttt{case}\ x_1\ \{.e_1\}\{x_2 x_3.e_2\} \longrightarrow_A m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, e_1}$$

DS-Case-PSet-Cons
$$\frac{(\{B\} \uplus p) = \dot{\gamma}(x_1)}{m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, \texttt{case}\ x_1\ \{.e_1\}\{x_2 x_3.e_2\} \longrightarrow_A m, \{x_2 \mapsto \{B\}, x_3 \mapsto p\} \uplus \dot{\gamma}, \dot{\delta}, \dot{\kappa}, e_2}$$

DS-Par
$$\frac{p = \dot{\gamma}(x) \qquad A \in p}{m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, \texttt{par}\ x\ e \longrightarrow_A m \cap p, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, e}$$

DS-ParEmpty
$$\frac{p = \dot{\gamma}(x) \qquad A \notin p \qquad \dot{\gamma}' = \{x' \mapsto \bigstar\} \uplus \dot{\gamma}}{m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, \texttt{par}\ x\ e \longrightarrow_A m, \dot{\gamma}', \dot{\delta}, \dot{\kappa}, x'}$$

DS-App
$$\frac{\dot{v}_1 = \dot{\gamma}(x_1) \qquad \dot{v}_2 = \dot{\gamma}(x_2) \qquad \langle \lambda_z x.\ e, \dot{\gamma}' \rangle = \dot{v}_1}{m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, x_1\ x_2 \longrightarrow_A m, \{z \mapsto \dot{v}_1, x \mapsto \dot{v}_2\} \uplus \dot{\gamma}', \dot{\delta}, \dot{\kappa}, e}$$

DS-LetPush
$$\frac{\dot{\kappa}' = \langle \texttt{let}\ x = \square\ \texttt{in}\ e_2 \mid m, \dot{\gamma} \rangle :: \dot{\kappa}}{m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow_A m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}', e_1}$$

DS-LetPop
$$\frac{\dot{\gamma} \vdash_m \dot{\delta}, a \hookrightarrow \dot{\delta}', \dot{v} \qquad \dot{\kappa} = \langle \texttt{let}\ x = \square\ \texttt{in}\ e \mid m', \dot{\gamma}' \rangle :: \dot{\kappa}'}{m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, a \longrightarrow_A m', \{x \mapsto \dot{v}\} \uplus \dot{\gamma}', \dot{\delta}', \dot{\kappa}', e}$$

$$\boxed{C \rightsquigarrow C}$$

DS-Step
$$\frac{\dot{\varsigma} \longrightarrow_A \dot{\varsigma}'}{\{A \mapsto \dot{\varsigma}\} \uplus C \rightsquigarrow \{A \mapsto \dot{\varsigma}'\} \uplus C}$$

DS-Share
$$\frac{\begin{array}{lll}
\texttt{share}[x_1 \rightarrow x_2]\ x_3 = C(m).e & \vdash_p \psi & C' = \{A \mapsto (m, \{x \mapsto \dot{v}\} \uplus \dot{\gamma}, \dot{\delta}, \dot{\kappa}, x) \\
\quad p = C(m).\dot{\gamma}(x_1) & m = C(m).m & \quad \mid C(A) = (m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, e), \\
\quad q = C(m).\dot{\gamma}(x_2) & m = p \cup q & \quad A \in q \implies \dot{v} = i^{\text{enc}\#q}, \\
\quad i^{\psi} = C(p).\dot{\gamma}(x_3) & q \neq \varnothing & \quad A \in p \land A \notin q \implies \dot{v} = \bigstar\}
\end{array}}{C_0 \uplus C \rightsquigarrow C_0 \uplus C'}$$

DS-Reveal
$$\frac{\begin{array}{lll}
\texttt{reveal}[x_1 \rightarrow x_2]\ x_3 = C(m).e & & C' = \{A \mapsto (m, \{x \mapsto \dot{v}\} \uplus \dot{\gamma}, \dot{\delta}, \dot{\kappa}, x) \\
\quad p = C(m).\dot{\gamma}(x_1) & m = C(m).m & \quad \mid C(A) = (m, \dot{\gamma}, \dot{\delta}, \dot{\kappa}, e), \\
\quad q = C(m).\dot{\gamma}(x_2) & m = p \cup q & \quad A \in q \implies \dot{v} = i, \\
\quad i^{\text{enc}\#p} = C(p).\dot{\gamma}(x_3) & q \neq \varnothing & \quad A \in p \land A \notin q \implies \dot{v} = \bigstar\}
\end{array}}{C_0 \uplus C \rightsquigarrow C_0 \uplus C'}$$

Figure 13 λ-Symphony distributed semantics (full figure).

About the authors

Ian Sweet ins@cs.umd.edu

David Darais darais@galois.com

David Heath heath.davidanthony@gatech.edu

William Harris wharris@galois.com

Ryan Estes restes@uvm.edu

Michael Hicks mwh@cs.umd.edu