Choreographies for Secure Computation

Manuel Veiga
Master in Information Security
Department of Computer Science
Faculty of Sciences of the University of Porto
2025

Choreographies for Secure Computation

Manuel Veiga
Dissertation carried out as part of the Master in Information Security
Department of Computer Science
2025

Supervisor
Hugo José Pereira Pacheco, Assistant Professor, Faculty of Sciences of the University of Porto

Acknowledgements

I want to start by sincerely thanking my advisor, Prof. Hugo Pacheco, for his patience, support, and kind sharing of knowledge during this thesis.

I am also grateful to the Department of Computer Science (DCC) and the Faculty of Sciences for making this work possible, particularly for the administrative support and the infrastructure that enabled the research.

I owe a great debt of gratitude to my family: my mother, father, and sister, for their unconditional support and encouragement, particularly in these difficult times. I would also like to thank my closest friends: Francisco Fernandes, Flávio Dantas, Bruno Coimbra, Isaac Silva, Pedro Santos, João Barbosa, Nuno Domingos, Hugo Almeida, Pedro Vilar, as well as everyone who supported me along the way.

Resumo

A programação coreográfica é um paradigma de programação para sistemas distribuídos, onde um único programa global, designado por coreografia, define as interações de todos os participantes na rede. A computação multi-parte segura (MPC) permite a um grupo de participantes mutuamente desconfiados o cálculo em conjunto de uma função arbitrária sobre os seus dados privados, sem revelar qualquer informação além do resultado da computação. Embora as linguagens MPC facilitem a construção de protocolos, muitas vezes não garantem coordenação e são frequentemente implementadas como linguagens autónomas, o que limita a sua integração em ecossistemas de programação mais vastos.

Nesta tese, apresentamos o ChoreoMPC, uma linguagem específica de domínio embebida em Haskell que combina as garantias de privacidade da MPC com as garantias de coordenação das coreografias. A nossa eDSL disponibiliza análogos seguros de operações aritméticas e lógicas sobre valores partilhados secretamente, através de uma interface independente do backend, descrita com classes de Haskell. Assim, o mesmo código de um protocolo pode ser executado em diferentes frameworks, sem necessidade de alterações, uma vez que a interface é instanciada para vários backends backends de MPC.

Usando esta interface, implementámos algoritmos complexos como o LWZ secure shuffling, assim como primitivas padrão de MPC, tais como partilha secreta de valores, multiplicação e comparação. Estes casos de estudo mostram que as abstrações coreográficas permitem capturar de forma natural e sucinta os padrões utilizados em MPC, e simultaneamente continuam a oferecer o potencial de expressar cenários de computação distribuída mais gerais. No geral, este estudo demonstra que a programação coreográfica oferece uma forma adaptável e intuitiva de expressar e executar computações multi-parte seguras.

Palavras-chave: Programação Coreográfica, Computação Multi-parte Segura, Sistemas Distribuídos, Ausência de Deadlocks, Partilha Secreta, Linguagens Específicas de Domínio, Haskell, Interface de Programação

Abstract

Choreographic programming is a paradigm for distributed systems where a single global program, known as a choreography, defines the interactions of all participants in the network. Secure multi-party computation (MPC) allows a group of mutually distrustful parties to compute a joint arbitrary function on their inputs without disclosing any information other than the result of the computation. Although MPC languages facilitate the construction of protocols, they often do not ensure coordination and are commonly implemented as standalone languages, thereby lacking integration with larger programming ecosystems.

In this thesis, we present ChoreoMPC, a Haskell embedded domain-specific language that combines the privacy guarantees of MPC with the coordination guarantees of choreographies. Our eDSL enables secure analogs of arithmetic and logical operations on secret-shared values using a backend-agnostic interface that is described with Haskell type classes. The same protocol code can be run across frameworks without requiring changes, as the interface is instantiated for multiple MPC backends.

Using this interface, we implemented complex algorithms like the secure LWZ shuffle as well as standard MPC primitives like secret sharing, multiplication, and comparison. These case studies demonstrate how choreographic abstractions encapsulate MPC patterns in a clear and concise manner, while simultaneously maintaining the ability to express broader distributed computing scenarios. All things considered, this study shows that choreographic programming offers a flexible and intuitive framework for expressing and carrying out safe multi-party computations.

Keywords: Choreographic Programming, Multi-party Computation, Distributed Systems, Deadlock Freedom, Secret Sharing, Domain-Specific Languages, Haskell, Programming Interface

Table of Contents

List of Tables

List of Figures

List of Abbreviations

API Application Programming Interface. 9

BGW Ben-Or–Goldwasser–Wigderson. 24

BMR Beaver–Micali–Rogaway. 28

CCD Chaum-Crépeau-Damgard. 24

DSL Domain-Specific Language. 27

eDSL Embedded Domain-Specific Language. 2

EPP EndPoint Projection. 4

GMW Goldreich–Micali–Wigderson. 24

KoC Knowledge of Choice. 12

MPC Secure Multi-party Computation. 1

OT Oblivious Transfer. 25

SIMD Single Instruction Multiple Data. 28

1. Introduction

Developing correct and secure distributed systems remains one of the most significant obstacles in modern software engineering. Traditional approaches require that developers manually coordinate the communication patterns while developing individual programs for each participant, which can be difficult to maintain and error-prone.

The difficulty becomes even greater in Secure Multi-party Computation (MPC) where many independent and mutually-distrusting parties collaborate to evaluate a joint function over their secret inputs while revealing nothing beyond the final output. In such scenarios, tiny, easily overlooked message coordination errors (such as mismatched sends and receives) can lead to race conditions, non-termination and deadlocks, as well as serious violations of privacy guarantees. These problems are made even more difficult by a broader set of challenges that arise when putting MPC protocols into practice. In many cases, protocol requirements are based on theoretical models that assume ideal conditions, such as perfectly timed execution by all parties and dependable communication routes. However, real-world systems rarely function in this manner: messages may arrive late, networks may be unstable, and various parties may not always be in sync. This disconnect between theory and reality can result in subtle and difficult-to-detect flaws that threaten both the correctness of the computation and the security assurances of the protocol.

Choreographic programming [1] is a programming paradigm that tackles this coordination problem by allowing programmers to describe the whole system from a global perspective, known as a choreography. A compilation phase known as endpoint projection then automatically generates process-local programs that run on each participant, assuring deadlock freedom by construction. In spite of this advantage, choreographic programming has received little study in the context of secure computation, which requires further security guarantees in addition to deadlock-freedom. MPC protocols, in particular, rely on strong privacy guarantees, which are frequently enforced in MPC programming languages utilizing type systems that distinguish between public and secret variables, ensuring that confidential information is never disclosed during computation or communication. This concept of privacy of values is naturally compatible with choreographic models in which the visibility and flow of values across participants can be made explicit. Additionally, secure computing often assumes adversarial models, including active adversaries who may unilaterally stray from the protocol and honest-but-curious adversaries who respect to

the protocol but attempt to deduce secret information from the messages they receive from other parties. Addressing the active adversarial model requires additional guarantees, such as, input independence (preventing corrupt parties from tailoring their inputs based on honest parties' secrets), guaranteed output delivery (ensuring honest parties always obtain their outputs), and fairness (ensuring corrupt parties receive their outputs only if honest parties do), which typically go beyond what standard choreographic frameworks provide. These features are outside the focus of this thesis, but they are relevant for future extensions.

This thesis explores the feasibility of using choreographic programming as a high-level abstraction for secure multiparty computation. We present an Embedded Domain-Specific Language (eDSL) in Haskell that combines cryptographic privacy and communication correctness into a single framework. In addition to highlighting the potential of this approach for expressing and reasoning about distributed protocols, our eDSL shows how choreographic principles can be extended to secure computation by offering a backend-agnostic interface instantiated for multiple MPC systems.

2. State of the Art

This chapter provides the theoretical foundations essential to fully understand the context and reasons for this research. It introduces the thesis's core concepts and technologies, starting with choreographies, choreographic programming, the paradigm that allows for the global specification of distributed systems, and progressing to MPC primitives and languages.

2.1. Choreographies

In computer science, the concept of choreographies existed before it was formally recognized as a paradigm for programming. It was first introduced by the World Wide Web Consortium's Web Services Choreography Description Language (WS-CDL) [2] as a way to write web services. The official specification [2] defines this language as follows: "an XML-based language that describes peer-to-peer collaborations of participants by defining, from a global viewpoint, their common and complementary observable behavior; where ordered message exchanges result in accomplishing a common business goal". In other words, WS-CDL formed a declarative, XML-based framework for expressing how many web services should interact, not from the standpoint of individual services, but from a global perspective that describes the sequence and structure of interactions among all participants.

This global description of the entire interaction introduced a new way of conceptualizing distributed systems: rather than specifying local behaviors independently at each endpoint, the system would be mapped from a global perspective, generating a coordinated interaction plan that governs the behavior of all participating processes. This shift in perspective ultimately led to the definition that we know today as a *choreography*. Montesi provides a concise formulation in his book [3]:

> In computer science, a choreography is a coordination plan for processes that participate in concurrent and distributed systems. It defines the communications that these processes are expected to perform, usually with the intention that these processes collaborate in order to achieve a joint goal.

2.2. Endpoint Projection

While choreographies give a global specification of distributed process interactions, these abstract descriptions are hard to map into concrete realizations. Qiu et al. [4] bridged this gap in their seminal work, Towards the Theoretical Foundation of Choreography, by exploring some elementary issues of implementing choreography such as semantics, projection, and creation of a simple WS-CDL choreography language, *Chor*, with a role-oriented process language.

To achieve this, the authors started by giving explicit syntactic and semantic underpinnings for both the choreography language and the role-oriented process language. Based on this basis, the authors explored the concept of projection, a mechanism for translating a global choreography into a set of local process specifications, each capturing the behavior of a single participant. To address recognized limits in this translation, such as maintaining proper interaction sequencing, they investigated and proposed several refined projection strategies.

However, the most significant contribution of their work was the formalization of the dominant role notion, which assigns a single participant to make decisions in control-flow scenarios such as branches and loops. Building upon this concept, they expanded both their choreography and process languages with the constructs of dominated choices (where the dominating role chooses the execution path to be followed by all participants) and dominated loops (where it dictates iteration behavior). These extensions were ultimately integrated into their main projection mechanism, which is an improvement on the natural projection, ensuring that all players remain synchronized and accurately follow the planned global interaction structure.

Subsequent studies aimed to offer more formal and trustworthy mechanisms for this translation, building on the design documents of WS-CDL [2] and the fundamental issues expressed by Qiu et al. [4] on the application of choreographies and the difficulties of converting global specifications into coherent local behaviors. Carbone et al. [5] made substantial progress toward formalizing this translation procedure. Their work, Structured Communication-Centered Programming for Web Services, proposed a full formal theory of EndPoint Projection (EPP), a technique for deriving each participant's local behavior directly from a global choreography, as illustrated in 2.1. To achieve this, Carbone et al. [5] defined two typed calculi: a global calculus, distilled from WS-CDL, designed to describe interaction scenarios from a global viewpoint; and an end-point calculus, based

on an applied π-calculus, that precisely identifies the local behavior of each participant. The main accomplishment of their work is the identification of three vital principles that contribute to a strong and comprehensive EPP:

- **Connectedness:** A basic local causality principle ensuring that the communication structure is well-formed.
- **Well-Threadedness:** A stronger locality principle based on session types, ensuring that interactions are properly sequenced and structured.
- **Coherence:** A consistency principle that guarantees each participant's behavior aligns with the global description.

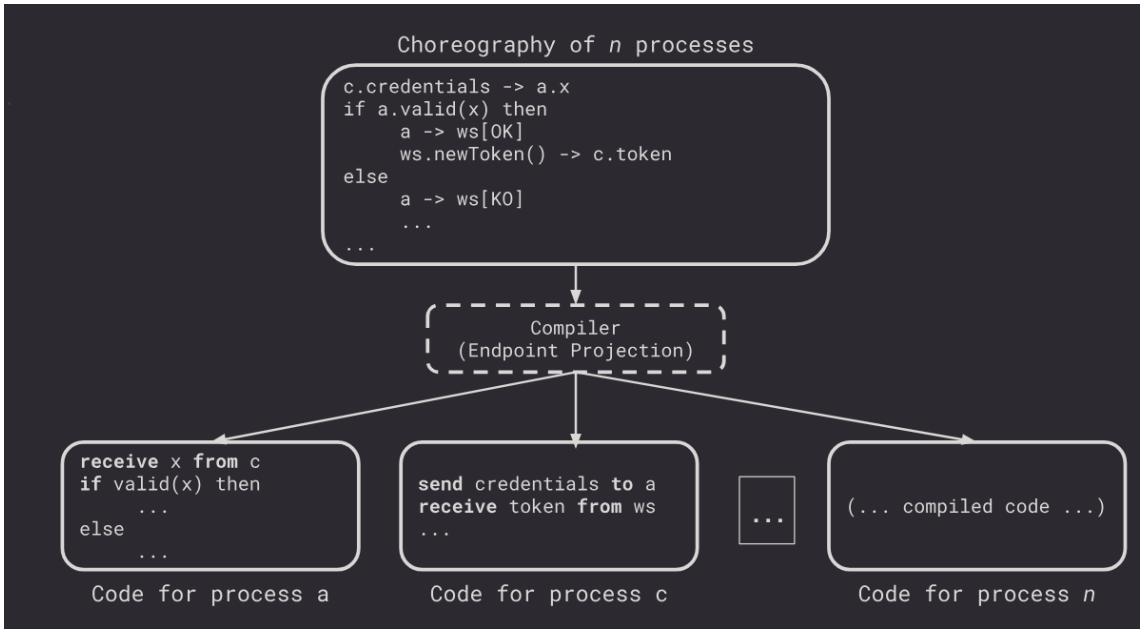

Fig 2.1 - Compilation of a Choreography [6]

These principles ensure that only the behaviors defined globally are realized as end-point communications, resulting in correct-by-construction and deadlock-free implementations. This method not only tackles the technical flaws found in previous models, but also sets up choreographic programming as a disciplined process for developing safe and dependable concurrent systems.

2.3. Choreographic Programming

Although choreographies have been shown to offer potential for developing safe distributed software, their actual application has been limited for years. Qiu et al. [4] and Carbone

et al. [5] pioneered EPP, which translates global interaction specifications into local implementations. However, despite this progress, choreographies at the time lacked key capabilities to construct scalable and fully functional distributed systems.

Essential features, such as asynchrony, mobility, compositionality, and multiparty session support, were either completely lacking or addressed only partially. For example, Carbone et al. [5] focused solely on binary sessions with only two participants. These limitations severely limited the practicality and expressiveness of choreographic techniques in real-world distributed software engineering.

A major turning point was reached in 2013 with Fabrizio Montesi's PhD thesis, Choreographic Programming [1], which began to address these challenges in a systematic way. Montesi's work established a new paradigm for programming, where programs are described as choreographies, which are high-level descriptions of the interactions between separate processes (also known as roles or participants).

To accomplish this, Montesi, during the development of his thesis, worked on three significant achievements that jointly advance the state of the art in choreographic programming:

1. **Formal Model and Type Theory**
   Montesi's first contribution involved creating a formal model and type theory for choreographic programming that allows for asynchronous, mobile, modular development and multi-party sessions. This approach guaranteed safety by connecting choreographies to distributed systems using a version of the π-calculus , which is the standard model for mobile processes. With this, the translation of the global specification into a collection of endpoint processes satisfies the desired safety aspects of choreographies, particularly deadlock freedom and the elimination of race conditions.

2. **Linear Connection Logic (LCL)**
   Montesi's second contribution introduced *Linear Connection Logic*, a logical framework based on extensions of Linear Logic, to precisely capture the semantics of choreographic interactions. Through a Curry–Howard correspondence, LCL established a direct relationship between proofs and choreographic programs. This enables *round-trip development*: choreographies can be systematically compiled into process calculus implementations, and conversely, process-level code can be abstracted back into choreographies, bridging specification and implementation.

3. **Chor Framework**

   Finally, to validate the feasibility of his theoretical work, Montesi developed a prototype Integrated Development Environment called Chor, not to be confused with the earlier language of the same name introduced by Qiu et al. [4]. Chor includes a type checker to verify protocol adherence and compiles choreographic programs into executable code using an extended version of the *Jolie* programming language [7], a general-purpose language for distributed computing, enriched to support asynchronous multi-party sessions [8].

Montesi's introduction of choreographic programming developed a fundamental paradigm in which global interaction specifications are directly compiled into distributed implementations that are correct by design. Following this fundamental work, researchers started exploring how choreographic solutions could be incorporated into conventional programming paradigms to increase their practical utility.

A prominent innovation in this field was **Choral** [9], an object-oriented choreographic programming language which utilizes choreographies to create modular Java libraries. Choreographies in Choral are represented as classes, and their instances are objects with state and behavior that several roles collaborate to accomplish. The integration of choreographic and object-oriented programming provides several major benefits. For instance, choreographies provide developers with high-level abstractions and static checks, allowing them to create more concise and correct concurrent and distributed object-oriented code. When combined with its object-oriented features, such as object passing, Choral grows into the first higher-order choreographic programming language, allowing choreographies to be parameterized over others without the need for central coordination, increasing the expressive power of choreographies while also making them more reusable and flexible. Thanks to its special blend of expressiveness, modularity, and compatibility with other languages, Choral is the first choreographic programming language that truly permits scalable and realistic software development.

2.4. Functional Choreographic Programming

2.4.1 Chorλ and Pirouette

Choral development represented a significant step forward in the growth of choreographic programming. Choral showed that it was possible to develop modular, distributed Java

libraries from high-level choreographies by including choreographic ideas into an object-oriented context. Choral increased the model's expressive capacity and modularity by introducing powerful characteristics such as completely distributed object passing and higher-order choreographies. Such enhancements, nevertheless, according to [10]:

> Choral deviated significantly from known theories of choreographies, and in particular introduced the possibility of expressing higher-order choreographies (choreographies parameterised over choreographies) that are fully distributed. As a consequence, it is unclear whether the usual guarantees of choreographic programming can still hold in the more general setting of higher-order choreographies.

To address such theoretical issues, Cruz-Filipe et al. [10] designed Chorλ, the first strategy that synthesized choreographic programming and functional programming. Chorλ was the first choreographic programming paradigm founded on λ-calculus. It had a well-founded theory for modeling sophisticated, modular choreography based on a well-established basis for higher-order programming. Another unique feature of Chorλ's was that, unlike Choral's object-oriented approach, which had accomplished communication among roles by method calls, Chorλ's functional term to carry out a communication was a function, and thus could be constructed from other terms as happens with functional programming.

Another similar, but different, progress in the field of functional choreographies was Pirouette [11], a language designed by Hirsch and Garg that puts a formally verified implementation foundation on the ground of Chorλ theory. Pirouette extended Chorλ's view of choreographic programming in the sense that it includes higher-order functional programming as an inherent part of choreographic constructs, whereas Chorλ re-interpretation of choreographic programming was founded on the λ-calculus viewpoint, with communication as a first-class function to enable higher-order abstraction and compositional reasoning. For this purpose, Pirouette presented a statically typed choreography language that is defined generically in terms of a local messaging language. The language ensures correctness by construction by enabling endpoint projection of centralized specifications into distributed executions. The type system is richly defined, and strong semantic correctness is ensured by mechanically proving safety guarantees, such as deadlock freedom, in Coq. Moreover, Pirouette supports higher-order choreographies, which are modular, reusable, and scalable distributed system architectures supported by feeding the choreography terms themselves as values. Together, these features make Pirouette a valuable addition

to the field, balancing expressive power and realistic verifiability in functional choreographic programming. One major limitation of Pirouette, however, is that it does not directly support application to standard development processes, something that Choral supports more directly.

2.4.2 HasChor

HasChor [12] provides a novel and practical architecture for functional choreographic programming, where choreographies are expressed as computations within a monad. Utilizing Haskell's expressive type system, tooling, and ecosystem, HasChor is implemented as an eDSL that enables programmers to create choreographic programs without the need for dedicated compilers or additional infrastructure.

HasChor presents an innovative and effective model for functional choreographic programming in which choreographies are represented as computations within a monad. The model supports cutting-edge features that promote modularity and code reuse, most notably, higher-order choreographies, where one choreography can be passed as an argument to another, and location-polymorphic choreographies, which allow choreographies to abstract over participants rather than being tied to fixed roles (such as ”buyer” or ”seller” in the bookseller protocol). For instance, a developer can create a generic choreography over a participant a, such as:

```
bookseller :: Proxy l -> Choreo IO (Maybe Day @ l)
```

which is generic over a participant `l`, and later instantiate `l` with a concrete node at runtime. In this approach, HasChor makes choreographic programming accessible to regular Haskell programmers, providing a seamless and reusable model based on standard functional programming methods. Similar to how Choral functions for Java, HasChor serves Haskell by integrating choreographic programming into the host language and providing a foundation for writing correct-by-construction distributed systems. According to its authors, HasChor is the first implementation of a functional choreographic programming model that is both monad-based and written in a general-purpose programming language, distinguishing it from previous proposals that were either purely theoretical or limited to proof-assistant prototypes. Along with the code for the bookseller protocol, written in HasChor's choreographic dialect and shown in Figure 2.2, the following section describes HasChor's Application Programming Interface (API), highlighting its fundamental structures, expressive potential, and current limitations in practical use.

HasChor API

As stated earlier, HasChor is an eDSL in Haskell, primarily centered around the `Choreo` monad. This monad, denoted as `Choreo m a`, represents a global choreography where the computations across all participants are defined, ultimately yielding a value of type `a`. The computations run within a base `monad m`, such as `IO`. Located values are expressed with the type `a @ l`, where `a` is the value's type and `l` is the participant (node) where this value stands.

The HasChor API primitives:

- `locally`: Performs local computations at a specified location.
- `(~>)`: Handles communication between locations.
- `cond`: Enables conditional branching between locations.

```
bookseller :: Proxy l -> Choreo IO (Maybe Day @ l)
bookseller someBuyer = do
  title <- someBuyer `locally` \_ -> do
      putStrLn "Enter the title of the book to buy"
      getLine
  title <- (someBuyer, title) ~> seller
  price <- (seller, \un -> return $ priceOf (un title)) ~~> someBuyer
  decision <- buyer `locally` \un -> return $ (un price) < budget
  cond (buyer, decision) \case
    True  -> do
      deliveryDate <- (seller, \un -> return $ deliveryDateOf (un title)) ~~
    > someBuyer
      someBuyer `locally` \un -> do
        putStrLn $ "The book will be delivered on " ++ show (un deliveryDate)
        return $ Just (un deliveryDate)
    False -> do
      someBuyer `locally` \_ -> do
        putStrLn "The book's price is out of the budget"
        return Nothing
```

Fig 2.2 - HasChor's implementation of the Bookseller Protocol [12, 13]

Local Computation:

The `locally` function performs local computations at a particular location. This function receives a location `l` and a local computation of type `m a`, returning a value of type `a` at `l`:

```
locally :: Proxy l -> (Unwrap l -> m a) -> Choreo m (a @ l)
```

Values at `l` can be unwrapped by the local computation using the unwrap function (in the second parameter of the `locally`), but not values located at any other locations. This function is simply an alias:

```
Unwrap l = forall a. a @ l -> a
```

For example, the seller in the bookseller choreography in [12] uses the following code to look up the book's pricing.

```
price <- seller `locally` \un -> return (priceOf (un t))
```

Once the book title has been unwrapped using `Unwrap l`, the seller calls `priceOf`, a lookup function that outputs the price of a book based on its title, and then returns its outcome.

Communication:

A choreographic language requires a way for participants to communicate with one another. The function `(~>)` returns a value of the same type `a` but located at the receiver after receiving a pair of the sender's location (`Proxy l`) and a value of the same type `a` situated at the sender (`a @ l`) and the respective receiver's location (`Proxy l'`):

```
(~>) :: (Proxy l, a @ l) -> Proxy l' -> Choreo m (a @ l')
```

Using the same bookseller choreography example, for instance, the seller uses the following code to communicate the price of the book that it locally computed to the buyer:

```
price' <- (seller, price) ~> buyer
```

Additionally, HasChor offers a derived operation `(~~>)` that combines `locally` and `(~>)` for convenience.

```
(~~>) :: (Proxy l, Unwrap l -> m a) -> Proxy l' -> Choreo m (a @ l')
```

Conditionals:

HasChor's conditionals embrace Knowledge of Choice (KoC) [3, 14], a unique property of choreographic languages. This ensures that when a choice is made between alternative behaviors, all impacted users are informed of the decision. For instance, the bookseller protocol requires the buyer to notify the seller of their intention to purchase or not. This would require the programmer to write a `(~>)` statement in each branch of the conditional in a choreography, which would be tiring. In order to avoid the requirement for explicit synchronization, HasChor features the `cond` function, a single conditional that broadcasts the choice from one node to all others automatically, as shown in Figure 2.3. This function takes a location (`Proxy l`), a branching value located at that participant (`a @ l`), and a function that describes the follow-up choreographies based on the branching value as inputs (`a -> Choreo m b`) and returns one of the choreographies.

```
cond :: (Proxy l, a @ l) -> (a -> Choreo m b) -> Choreo m b
```

Using the same bookseller choreography example yet again, the customer is presented with the book's pricing and is then given the option to purchase the book. If so, the seller calculates the book's delivery date and notifies the customer by producing a date; if not, nothing occurs.

```
1  cond (buyer, decision) \case
2    True  -> do
3      deliveryDate <- (seller, \un -> return $ deliveryDateOf (un title)) ~~>
       someBuyer
4      someBuyer `locally` \un -> do
5        putStrLn $ "The book will be delivered on " ++ show (un deliveryDate)
6        return $ Just (un deliveryDate)
7    False -> do
8      someBuyer `locally` \_ -> do
9        putStrLn "The book's price is out of the budget"
10       return Nothing
```

Fig 2.3 - HasChor's cond operator [12]

With this, HasChor marks a major advancement in choreography programming. The framework supports higher-order and location-polymorphic choreographies and provides free access to all of Haskell's libraries, tooling, and rich ecosystem. However, according to [15], although having an intuitive and easily understandable API based on monads freer

monads, it has a number of shortcomings that restrict its usefulness in larger or more realistic contexts. These limitations include the inability to enumerate the participants in a choreography, which prevents the implementation of algorithms that abstract over their number of participants, and the low efficiency of conditionals because it broadcasts the value of the scrutinee of a conditional expression to all parties in a choreography, regardless of where that information is actually needed. The following section introduces MultiChor, a language that directly tackles these issues by embracing census polymorphism while also boosting choreographic expressiveness, scalability, and efficiency.

2.4.3 MultiChor

Motivated by HasChor's limitations, Bates et al. introduced MultiChor [16], a new Haskell library for choreographic programming, with the goal of filling two significant gaps in previous work: (i) the inefficiency resulting from unnecessary communication in conditional expressions, and (ii) the lack of support for census polymorphism, that is, choreographies parameterized not only by the number but also by the identities of participants. The term "census" refers to the set of parties involved in a choreography, and although MultiChor is not the first language to track censuses (languages such as Chorλ [10] and Pirouette [11] incorporate this concept into their type systems), it distinguishes itself by making the census an explicit, type-level parameter.

As illustrated in Figure 2.2, HasChor supports polymorphism over multiple locations by passing one value-level proxy per role, for instance, `Proxy l -> Proxy l'` for two roles, or `Proxy l -> Proxy l' -> Proxy l''` for three roles, and so on. However, this approach commits the choreography to a fixed arity at definition time: adding or removing roles changes the type and the function's parameter list. In other words, while HasChor can be polymorphic over which parties fill those roles, it is not polymorphic over how many roles there are. Consequently, HasChor cannot express choreographies that are parameterized by the number of participants, which makes it impractical to define general-purpose protocols (such as MPC) that should work with an arbitrary number of parties. Supporting census polymorphism is therefore one of the primary architectural distinctions between MultiChor and HasChor, leading to changes in the specification of the `Choreo` monad: in MultiChor, the census is now the first type argument.

Apart from this, MultiChor combines several concepts from other choreographic languages, most notably from He-λsmall [17], a language that presents a KoC strategy based on

multiply-located values, a single value that numerous participants jointly know. He-λsmall makes use of case expressions for conditional branching, which limits the census inside each branch to parties that are knowledgeable of the guard value. In terms of communication, it uses a multicast operator, where `com` implies a message that is being sent from one participant to an array of recipients.

Based on these ideas, Bates and Near [17] showed that He-λsmall can theoretically achieve the same level of communication efficiency as modern select-and-merge KoC techniques, such as those in Chorλ [10]. With a number of illustrative examples, they demonstrate the expressiveness and ergonomics of multi-local and multicast choreographic programming, providing a convincing third paradigm that lies between the broadcast-based strategy of HasChor and the select-and-merge models found in other functional choreographic systems. By integrating multiply-located values and an optimized KoC strategy, MultiChor directly overcomes the inefficiencies of HasChor's conditionals. Furthermore, by encoding the census as a type-level variable in Haskell, it provides full polymorphism in terms of both participant set size and membership.

The following section provides a description of MultiChor's API, highlighting its relation to HasChor's, fundamental structures and features.

MultiChor API

According to Bates et al. [16]

> HasChor has a blunter API for writing choreographies, but as an eDSL it can apply Haskell's polymorphism "for free". MultiChor is a similar eDSL, with some of HasChor's skeleton intact at its heart, and can similarly apply Haskell's polymorphism to its lists of parties.

however, to achieve the purposed Location-set and census polymorphism, they needed to change the previous implementation of `Located` values. In order to represent a multiply-located value, the authors defined a new datatype:

```
Located (ps :: [LocTy]) a
```

which means a single value of type `a` that's known simultaneously by all participants in the list `ps`. the `LocTy` type is an alias for `Symbol`, representing unique participant identifiers at the type level, making `[LocTy]` a type-level list of participants, which is the census for that value.

The next step toward useful location-set polymorphism was the definition of a data type analogous to multiply-located values, but without the guarantee that the parties' respective values will all the the same. Based on this definition, the `Faceted` datatype was created, and analogously to `Located` values,

```
Faceted (ps :: [LocTy]) a
```

represents a value of type `a`, that could and most likely would be different, between all participants in the list `ps`. For instance, consider the value `Faceted '[Alice, Bob] Bool`. The boolean value located at Alice and Bob, could be both `True` or `False`, or different from one another. However, according to Bates et al. [16], `Located` and `Faceted` values by themselves do not cover certain communication patterns commonly needed in location-set polymorphic choreographies. In particular, it is commonly necessary for a party to transmit different values to each participant in a polymorphic list, or for the members of such a list to exchange messages with one another.

To accomplish this, the authors defined two primitives `fanOut` and `fanIn`. The `fanOut` primitive executes a choreography once for each participant in a list of parties `rs` requiring each run to return a value at that participant, resulting in a `Faceted rs a`. On the other hand, `fanIn` also executes a choreography for each participant in a list of senders `ss`, but requires each run to return a value located at a fixed recipient set `rs`, resulting in a `Located rs (Quire ss a)`, where a `Quire ss a` is simply a collection of one value of type `a` for each sender in `ss`.

With this new datatypes and `fanIn` and `fanOut` primitives, the Multichor API can be defined as follows:

A choreography monad `Choreo ps m a`:

- `ps` is a type-level list of parties participating in the choreography,
- `m` is an monad used to write local actions those parties can take (often this is simply IO),
- `a` is the returned value, typically this will be a `Located` or `Faceted` value.

Monadic primitives:

- `parallel` runs a local monadic computation in parallel across a list of parties `ls`. Like HasChor, this computation can use a provided function to unwrap `Located` and `Faceted` values, returning `Faceted ls a`.
- `congruently` runs the same pure computation across all `ls`, yielding `Located ls a`.
- `~>` communication between a sender and a list of receivers, producing `Located ls' a`.
- `locally` performs a local computation at a given location, yielding `Located '[l] a`.
- `enclave` embeds a choreography scoped to a smaller census into a larger one.
- `enclaveTo` Wrapper around `enclave` + `flatten`: run a sub-choreography over a smaller census and immediately re-locate its result to specified recipients in the outer census.
- `naked` unwraps `Located allparties a` to produce a plain `a`; useful for branching without hidden communication.
- `broadcast` sends a value from a sender to the entire census `ps`; the received value is then unwrapped by `naked`.
- `fanOut` performs a choreography once for each party in a list, returning a `Faceted` that gives each party its result.
- `fanIn` (Dual of `fanOut`) collects values from many parties and delivers them to a fixed recipient set, returning `Located Quire`.
- `scatter` distributes the elements of a `Quire` from a single owner to the respective parties, yielding a `Faceted`.
- `gather` collects values from a `Faceted` across many owners into a `Quire` located at a recipient set.

Pure operations for located values:

- `flatten` unwraps a nested `Located` value `(Located ms (Located ns a))` into a flat one `(Located ls a)`, where `ls` = `ms` ∩ `ns`. Only parties in both `ms` and `ns` will see the final value.

- `othersForget` casts a `Located` value to a smaller ownership set; useful when working with functions who's arguments have explicit ownership sets.
- `Unwrap / Unwraps` Functions given to `locally / parallel` to extract the part of a `Located / Faceted` value visible to a party or set of parties

In addition to `Faceted / Located` values, MultiChor relies heavily on type-level proofs to ensure that choreographies are well-formed. These proofs are evidence carried at the type level that certain properties about the participants' membership hold. In particular, the proof object `Member p ps` certifies that a party `p` is an element of the census `ps`, while `Subset xs ys` certifies that all parties in the list `xs` are contained in the larger census `ys`. These proofs are provided as parameters to many API functions in order to statically guarantee that only the intended participants take part in each communication. To make such proofs usable in practice, MultiChor includes a small collection of axioms and constructors for building them.

Axioms (constructors) for building type-level proofs:

- `(@@) :: Member x ys -> Subset xs ys -> Subset (x ': xs) ys` extends a subset proof by consing a new element at the front, given evidence that the element is already a member of the superset.
- `nobody :: Subset '[] ys` The empty-subset proof: shows that `[]` is always a subset of any list, often used as a base case when building larger subset proofs when using `(@@)`.
- `singleton :: forall p. Member p (p ': '[])` polymorphic proof that a party is a member of a singleton list.
- `allOf :: forall ps. Subset ps ps` proof that "everyone in the current census participates".
- `consSuper :: forall xs ys y. Subset xs ys -> Subset xs (y ': ys)` lifts a subset proof into a larger census.
- `listedFirst :: forall p1 ps. Member p1 (p1 ': ps)` proof/witness for "the first party" in a census.
- `listedSecond :: forall p2 p1 ps. Member p2 (p1 ': p2 ': ps)` proof/witness for "the second party" in a census.

- `inSuper` casts a `Member x xs` proof into `Member x ys` given `Subset xs ys`.

One of MultiChor's unique characteristics is that a lot of its abstractions rely on proof objects of a certain kind, such as `Member` and `Subset`, to confirm that a party is a member of a census or that one census is contained within another. The Ghosts of Departed Proofs technique [18] enables these by introducing the logical invariants at the type level and removing them at runtime. Such proofs serve two purposes in MultiChor: they serve as proof of the well-typedness of operations and also serve as names that are proxies for the type-level principals they represent. This implies that type-level information can be securely included into term-level code without sacrificing soundness, as the compiler will guarantee features like membership and subset relations. It also clarifies why constraints on proofs are used throughout MultiChor's API, where they fulfill both the roles of references to parties in choreographies and static guarantees.

MultiChor's implementation of the Bookseller protocol, depicted in Figure 2.4, illustrates these concepts The differences are minor but instructive when compared to the HasChor version in Figure 2.2. In MultiChor, the buyer role is passed as a parameter of type `Member a ps`, representing an arbitrary participant within the coalition `ps` and directly making use of the proof objects previously mentioned, which contrasts with HasChor, where the buyer is introduced through a value-level `Proxy l`. Furthermore, HasChor uses the more general `cond` primitive, while the buyer uses the dedicated `broadcast` primitive to determine whether the book is affordable.

```
1  bookseller :: Member a ps -> Choreo ("seller" ': ps) (CLI m) ()
2  bookseller someBuyer = do
3    let theBuyer = inSuper consSet someBuyer
4    title <- (theBuyer, getstr "Enter the title of the book to buy") -~> seller @@ nobody
5    price <- (seller, \un -> return $ priceOf (un seller title)) ~~> theBuyer @@ nobody
6    inBuyerBudget <- theBuyer `locally` (\un -> return $ un singleton price <= un ↩
   ↪ singleton budget)
7    broadcast (theBuyer, inBuyerBudget) >>= \case
8      True -> do
9        deliveryDate <- (seller, \un -> return $ deliveryDateOf (un seller title)) ~~> ↩
   ↪ theBuyer @@ nobody
10
11       theBuyer `locally_` \un -> putOutput "The book will be delivered on:" $ un ↩
   ↪ singleton deliveryDate
12     False -> do
13       theBuyer `_locally_` putNote "The book's price is out of the budget"
```

Fig 2.4 - MultiChor's implementation of the Bookseller Protocol [16, 19]

The relative simplicity of the Bookseller protocol itself is reflected in the structural similarity between the two versions despite these improvements. The abstractions provided by

MultiChor are fully utilized in more complex choreographies. To illustrate this, we present a second example, the Card Game from MultiChor's paper [16], depicted in Figure 2.5, displaying polymorphism between player sets, coordinated choices, and enclaves.

```
game = do
    let players = consSuper (allOf @players)
        dealer = listedFirst @"dealer"
    hand1 <- fanOut players \player -> do
        card1 <- dealer `_locally` getInput ("Enter random card for " ++ toLocTm player)
        (dealer, card1) ~> inSuper players player @@ nobody
    onTheTable <- fanIn players players \player -> do
        (player, players, hand1) ~> players
    wantsNextCard <- players `parallel` \player un -> do
        putNote $ "My first card is: " ++ show (un player hand1)
        putNote $ "Cards on the table: " ++ show (un player onTheTable)
        getInput "I'll ask for another? [True/False]"
    hand2 <- fanOut players \(player :: Member player players) -> do
        (dealer @@ inSuper players player @@ nobody `enclaveTo` listedSecond @@ ↩
↪ nobody) do
            choice <- broadcast (listedSecond @player, (player, wantsNextCard))
            if choice then do
                cd2 <- dealer `_locally` getInput (toLocTm player ++ "'s second card:")
                card2 <- (dealer, cd2) ~> listedSecond @@ nobody
                listedSecond `locally` \un -> pure [un player hand1, un singleton card2]
            else listedSecond `locally` \un -> pure [un player hand1]
    tblCrd <- dealer `_locally` getInput "Enter a single card for everyone:"
    tableCard <- (dealer, tblCrd) ~> players
    players `parallel_` \player un -> do
        let hand = un player tableCard : un player hand2
        putNote $ "My hand: " ++ show hand
        putOutput "My win result:" $ sum hand > card 19
```

Fig 2.5 - Card Game choreography [16, 20]

This choreography models a basic blackjack game where each player receives an initial card from the dealer. After that, each player is free to choose whether to request a second card. Finally, the dealer reveals a shared card that applies to everyone. Each player individually wins if the sum of their cards (modulo 21) is greater than 19. The implementation makes extensive use of MultiChor's API: the function is polymorphic over an arbitrary set of players, and the dealer is explicitly declared using `listedFirst`; the operators `fanOut` and `fanIn` spread out and collect values across players so that the dealer can deal cards and gather table information. The `parallel` combinator gives each player independent action when selecting a second card. Advanced coordination is stated by `enclaveTo`, which distributes control to a subset of parties, and `broadcast`, which assures consistency of choices within the enclave. Finally, input and output in this example are handled through a lightweight `CLI` monad included in MultiChor's repository specifically for interactive choreographies. Players use `putNote` and `putOutput` to display their hand and outcome, while the dealer employs `locally` to introduce external inputs

such as random cards. This IO mechanism is specific to the example implementation and mainly illustrates how choreographies can interact with user input and output.

2.5. Secure multi-party computation

Secure multi-party computation is a family of cryptographic protocols that enable many parties to compute a function over their individual inputs without disclosing those inputs to one another. However, unlike traditional cryptography, which ensures the security and integrity of communication, this model is responsible for protecting the privacy of each party's input while still allowing the joint computation to succeed. The following subsection describes the security model used to formalize MPC, explains how different classes of adversaries are captured in this model, and then highlights some of the basic primitives and classic protocols that support our work.

2.5.1 Security & Threat models

However, the term ”secure” in Secure Multi-Party Computation is not self-explanatory. It is too vague to be useful by itself, since it does not specify what guarantees are actually being asked of a protocol or how precisely such guarantees might be rigorously proven. A more technical definition of security is therefore required, one that explicitly lists the properties MPC is meant to provide and in which these properties can be proven.

An ad-hoc approach to defining security in this context is the ”laundry list” method, as described in [21], which involves enumerating specific conditions that would constitute a security violation. For example, such a list may contain ”rules” such as ”the adversary should not be able to learn a certain predicate of another party's input.” However, this method is time-consuming and prone to errors, since it's not trivial when the list could be considered complete. To circumvent such an error, the real-ideal paradigm was developed to define security in a precise, unambiguous, and provable way. This paradigm introduces an ”ideal world” and a ”real world” as reference models:

In the **Ideal World**, to compute an arbitrary function $\mathcal{F}$ securely, each party P_i privately sends its input x_i to a trusted third party $\mathcal{T}$. The trusted party then computes the desired function on all inputs,

$$(y_1, \ldots, y_n) = \mathcal{F}(x_1, \ldots, x_n),$$

and returns the corresponding output y_i to each party P_i. The entity $\mathcal{T}$ is referred to as a *trusted party* because, in this idealized model, an adversary may corrupt any of the

participants P_i but is assumed to have no control over $\mathcal{T}$, resulting in a secure execution by definition.

In the **Real World**, there is no trusted party, and instead to compute the same arbitrary function $\mathcal{F}$ must be computed by the parties themselves through a distributed protocol π. Analogously to the ideal world scenario, the adversaries may corrupt some parties. A protocol is then considered to be secure if its real-world execution is indistinguishable from the ideal execution, meaning that any action an adversary can perform in the real world could also be simulated in the ideal world.

So far, We have already mentioned that corruption can happen in both ideal and real world scenarios, but we have not defined yet how corrupted parties behave. To address this, the following paragraphs will introduce two standard adversarial models: Semi-Honest Security and Malicious Security, and address how they differ in adversarial power. These models are not related to the real/ideal definition of security, but they do influence the complexity of the protocols and the difficulty of proving their security within this paradigm.

A. Semi-Honest Security In the semi-honest security model, corrupt parties follow the protocol honestly while attempting to learn as much as possible from the messages they receive from other parties. This model is additionally frequently referred to as passive security, since semi-honest adversaries cannot take any actions other than attempting to gather private information by attending the execution of a protocol. When an adversary takes control of an honest party, its *view* consists of that party's private input, its internal randomness, and all messages it receives throughout the execution and in cases where multiple parties are corrupted simultaneously, the adversary's view is defined as the combined views of all corrupted parties. Since the adversary cannot tamper with the protocol, its entire attack is determined by its view, namely, its private input, internal randomness, and all messages it receives. In other words, any action the adversary can take can be reduced to whatever it can infer from this view.

Employing the Real-Ideal paradigm in this context, a protocol must be indistinguishable between the adversary's real-world view and what it might yield in the ideal world, where the only information available consists solely of the inputs given to the trusted party and the corresponding outputs, in order to be considered secure. This gives rise to the idea of a simulator, which is an algorithm that can create a simulated view that is indistinguishable from a real execution when given only the ideal-world inputs and outputs of the corrupted

parties. The presence of such a simulator proves that the actual protocol leaks no more information than would be revealed in an ideal world, providing security against semi-honest adversaries.

B. Malicious Security In contrast to the prior model, in the malicious security model, attackers can cause corrupted parties to diverge arbitrarily from the protocol design in order to violate security, more specifically they have the same ability to analyze protocol execution as a semi-honest adversary, but can also take any actions throughout the process, such as control, modify, and inject messages on the network.

Due to this increased adversarial power, applying the Real-Ideal Paradigm in the malicious setting introduces several additional complexities. Firstly, since these adversaries can tamper with the protocol it may not only affect privacy of the parties but also correctness of the results, altering the outputs of honest parties in ways that do not occur in the ideal world. Second, the malicious parties are not required to follow the protocol with a certain input, therefore it is unclear what inputs to provide them in the ideal scenario. To address this, it is acceptable for the simulator to select suitable inputs in place of the corrupted parties, a procedure called *input extraction*. Intuitively, in a secure protocol, anything the attacker can accomplish in the real world must be imitated in the ideal world by an optimal selection of inputs to the corrupted parties.

These challenges make the malicious model significantly more complex, since protocols must be equipped with additional cryptographic tests to ensure that deviations from honest behavior are detected. From a high-level perspective, active secure protocols can be considered semi-honest protocols enhanced with stronger primitives, such as consistency tests or zero-knowledge proofs, which prevent cheating by adversaries. In other words, the protocol's structure stays the same, and only the cryptographic building blocks differ. For simplicity's sake, the present thesis is focused on the semi-honest model, but our work could equally well include actively secure backends.

2.5.2 Basic primitives

This subsection presents basic cryptographic primitives used in MPC. While many such primitives exist, here we focus on secret sharing, which is central to the protocols considered in this thesis.

Secret Sharing As stated earlier by the Real-Ideal paradigm, in the real world, all parties involved in a secure protocol must communicate with one another, rather than relying on a trusted third party, while assuring the privacy of the inputs and the correctness of the result of the protocol. This necessity gave rise to an essential primitive known as Secret Sharing that sits at the core of many MPC approaches. Informally, a secret sharing scheme works as follows: Given a secret s, a (t, n)-secret sharing scheme splits s into n shares, such that any $t-1$ of the shares reveal no information about s, while any t shares allow complete reconstruction of the secret. There exists a multitude of secret sharing schemes. Following the presentation in [21, 22], we use the standard definition of a (t, n)-secret sharing scheme where:

A (t,n)-secret sharing scheme is defined as a pair of algorithms (Shr, Rec), where Shr is a (possibly randomized) sharing algorithm, and Rec is a reconstruction algorithm. Formally, let D denote the domain of secrets and D_1 the domain of shares. Then, the sharing algorithm is a function $Shr : D \rightarrow D_1$, and the reconstruction algorithm is a function $Rec : D_1^k \rightarrow D$, where $k \leq n$. Finally, this pair of algorithms (Shr, Rec) must satisfy the following properties:

- *Correctness*. Let $(s1, s2, \ldots, sn) = Shr(s)$. Then, for any subset of at least t shares, the reconstruction algorithm correctly recovers the original secret with probability 1: $Pr[\forall k \geq t, Rec(s_{i_1}, \ldots, s_{i_k}) = s] = 1$.
- *Perfect Privacy*. Any set of fewer than t shares reveals no information about the secret.

2.5.3 Classic MPC Protocols Overview

This subsection provides a brief overview of a foundational MPC protocol that has had a substantial impact on the field. Many of the languages and compilers addressed later still use the same underlying computation models, therefore this protocol remains very significant.

Multi-party circuit-based protocols This section discusses one major milestone in the development of MPC that followed Yao's initial concept of Garbled Circuits [23]. Unlike Yao's two-party approach, these protocols allow for the secure evaluation of functions by any number of parties, by using a linear secret sharing scheme, while also using a circuit representation of the computation where the answer is computed gate-by-gate. The

Goldreich–Micali–Wigderson (GMW) [24], Ben-Or–Goldwasser–Wigderson (BGW) [25], and Chaum-Crépeau-Damgard (CCD) [26] protocols make significant contributions. While all three represent substantial advances in the literature, the discussion will primarily center on the GMW protocol and its specification.

The GMW protocol is compatible with both Boolean and arithmetic circuit representations. To simplify the explanation, the Boolean version is initially described in the two-party scenario, followed by a discussion of how the protocol can be generalized to an arbitrary number of parties, adapted to arithmetic circuits, and important optimizations.

Assuming two parties $\mathcal{P}_1$ and $\mathcal{P}_2$ with inputs x and y, respectively, the protocol proceeds as follows: For each input bit x_i of $x \in \{0,1\}^n$, $\mathcal{P}_1$ generates a random bit r_i and sends it to $\mathcal{P}_2$. Next, $\mathcal{P}_1$ obtains a secret sharing of each x_i among both by setting its share to be $x_i^1 = x_i \oplus r_i$. $\mathcal{P}_2$, symmetrically, does exactly the same as $\mathcal{P}_1$, obtains a secret sharing of each y_i among both ($\mathcal{P}_1$ has (x^1, y^1) and $\mathcal{P}_2$ has (x^2, y^2), where $x = x^1 \oplus x^2$ and $y = y^1 \oplus y^2$). Once both inputs are secret shared, both parties can then evaluate the circuit gate-by-gate, assuming that the circuit consists of NOT, XOR, and AND gates. Evaluating NOT and XOR gates requires no communication, since a NOT gate is evaluated by flipping the party's share of the wire value, and XOR gates are evaluated by each party xor-ing the shares they already hold, as illustrated in Figure 2.6.

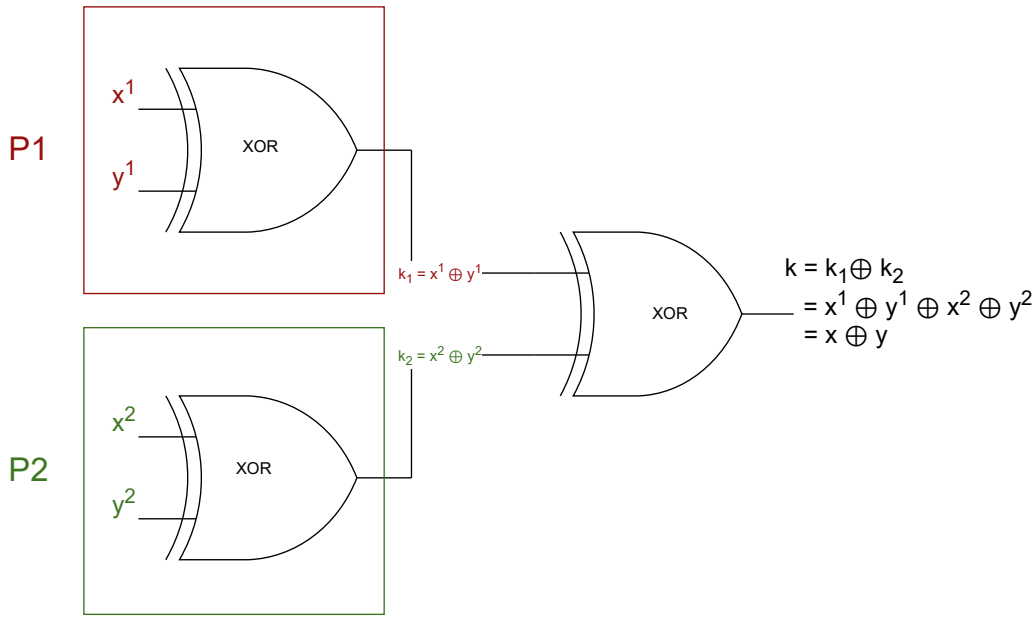

Fig 2.6 - XOR gate in two-party GMW (Adapted from [27])

However, evaluating AND gates is a bit more complicated. The expected output is:

$$(x \wedge y) = (x^1 \oplus x^2) \wedge (y^1 \oplus y^2),$$

and when applying the distributive property it yields:

$$(x^1 \oplus x^2) \wedge (y^1 \oplus y^2) = (x^1 \wedge y^1) \oplus (x^1 \wedge y^2) \oplus (x^2 \wedge y^1) \oplus (x^2 \wedge y^2).$$

The terms $x^1 \wedge y^1$ and $x^2 \wedge y^2$ can be computed locally by $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively, where, in contrast, the cross terms $x^1 \wedge y^2$ and $x^2 \wedge y^1$ require communication. These are then evaluated using a 1-out-of-4 Oblivious Transfer (OT): from $\mathcal{P}_1$'s perspective, its shares are fixed, while $\mathcal{P}_2$ holds two Boolean input shares. This leads to four possible combinations, as shown in Table 2.1.

Known		**Possibilities**		**Possible Outcomes** S
x_1	y_1	x_2	y_2	$(x_1 \wedge y_2) \oplus (x_2 \wedge y_1)$
0	1	0	0	0
		0	1	0
		1	0	1
		1	1	1

Tab 2.1 - Potential outcomes given $x_1 = 0$ and $y_1 = 1$ (Adapted from [27])

Next, $\mathcal{P}_1$ chooses a random mask bit $r \in \{0,1\}$ and creates the OT secrets table, by encrypting every "possible outcome" with $\oplus r$, as shown in Equation 2.5.3.

$$\begin{pmatrix} r \oplus S(0,0) \\ r \oplus S(0,1) \\ r \oplus S(1,0) \\ r \oplus S(1,1) \end{pmatrix}$$

Subsequently, $\mathcal{P}_1$ and $\mathcal{P}_2$ execute the OT protocol, with $\mathcal{P}_1$ acting as the Sender and $\mathcal{P}_2$ as the receiver. $\mathcal{P}_1$ organizes the table as four rows, each corresponding to a possible input combination and $\mathcal{P}_2$ then uses its two input bits (x_2, y_2) as selection indices to pick the appropriate row. After this, $\mathcal{P}_1$ keeps the random value r as its part of the gate's output

wire, and $\mathcal{P}_2$ gets the chosen value from the OT execution, and due to how the OT inputs are set up, both parties obtain a correct secret sharing of the gate's output wire, while simultaneously guaranteeing that neither party learns anything about the other's inputs or any intermediate values of the computation.

For the n-party Boolean GMW protocol, the core principles remain similar, but the execution is adapted to support secure computation among multiple parties. As before, when a party $\mathcal{P}_i$ secret-shares its input x_i, it needs to create $n-1$ random bits, sends them to the other parties and subsequently create its own share such that $r_i = x_i \oplus \bigoplus_{\substack{j=1 \\ j \neq i}}^{n} r_j$. The gate evaluation process remains the same for the XOR and NOT gates, since the parties can locally add their shares, and the AND gate can be generalized as shown in [21]:

$$\begin{aligned} c = a \wedge b &= (a_1 \oplus \cdots \oplus a_n) \wedge (b_1 \oplus \cdots \oplus b_n) \\ &= \left(\bigoplus_{i=1}^{n} a_i \wedge b_i \right) \oplus \left(\bigoplus_{i \neq j} a_i \wedge b_j \right) \end{aligned}$$

Now, for the arithmetic variant of the GMW protocol, the circuit is expressed using addition and multiplication gates instead of XOR and AND gates. While Boolean GMW relies on XOR-based secret sharing, where a value is split as $s = s_1 \oplus s_2 \oplus \cdots \oplus s_n$, arithmetic GMW uses additive secret sharing over a finite field, typically sharing a value ss as $s = s_1 + s_2 + \cdots + s_n$. As a result, addition gates become ”free” operations that each party can compute locally, similarly to XOR and NOT gates in Boolean setting, however, multiplication gates in arithmetic circuits introduce complexity by requiring communication between the parties, much like AND gates do in the Boolean version.

MPC protocols are typically not very fast, as their primary goal is to ensure strong privacy guarantees for all participants. These guarantees often rely on computationally expensive public-key primitives, such as OT. To improve efficiency, practical implementations of MPC protocols commonly adopt a two-phase approach: an offline phase and an online phase, where in the offline phase the parties precompute correlated randomness that will later serve as masks during the online phase. In the case of the Boolean GMW protocol, this typically involves the use of OT extension to generate a large batch of random OT correlations, in other words, pairs of correlated random values shared between two parties, where the receiver learns one value based on a private choice bit, and the sender remains oblivious to the choice. When it comes to the Arithmetic GMW protocol, it typically generates Beaver Multiplication Triples [28], which are shares (a, b, c) where

$c = a * b$, allowing for secure multiplication over a finite field.

2.5.4 MPC frameworks

As discussed in the previous chapter, general-purpose MPC protocols have existed in theory since 1980 [23, 24] and have shown great promise for practical application in fields such as statistical analysis, data mining, medical field research, etc. However, because of the efficiency of the underlying protocols, their adoption has been limited. Only recently have cryptographers made efforts to come up with solutions, such as creating highly-optimized special-purpose protocols for specific use-cases (though these lacked generalization and scalability) and the idea of building general-purpose MPC compilers that translate higher-level code into secure execution, such as secret-sharing backends or garbled circuits. Because such compilers automatically convert high-level code into the corresponding secure implementation while hiding the complex cryptographic details from the programmer, these compilers allow non-experts to write secure applications without having to deal directly with low-level protocol design.

Designing and implementing MPC compilers is challenging due to the need for a highly optimized compiler and crypto back-end, as well as a high-level MPC language that is expressive, flexible, and easy to understand for non-experts. Despite this, the feasibility of this approach was demonstrated by early instances, such as Fairplay [29], VIFF [30] and SEPIA [31], which created what is now a thriving field of research in MPC compiler design and implementation. As a result of this progress, many compilers and accompanying frameworks have been created, each with unique architectural decisions that affect their usability, performance, and applicability for different application domains. EMP-toolkit [32], Obliv-C [33], and TinyGarble [34], for example, rely on garbled circuits, whereas Sharemind [35] and PICCO [36] use a more flexible hybrid model that allows computations to be decomposed into multiple sub-protocols, each optimized for specific operations.

Among the different frameworks, Sharemind [35], Motion [37], and Symphony [38] stand out due to their unique design philosophies and reliance on additive linear secret sharing: Sharemind as an industrial platform with a fixed three-party model, Motion as a modular framework that supports an arbitrary number of parties and Symphony is a language-level Domain-Specific Language (DSL) that emphasizes coordination and expressiveness.

A. Sharemind Sharemind [35] was one of the first and most widely recognized frameworks for multi-party computation. It is based on additive secret sharing over the ring of 32-bit integers $\mathbb{Z}_{2^{32}}$, adopts the honest-but-curious adversary paradigm and fixes the computation model to exactly three computing parties. Security relies on an honest majority assumption, meaning that at least two out of the three parties must behave correctly. This design strikes a balance between security and efficiency, since more parties would increase fault tolerance, but also communication complexity.

This fixed three-party design reduces the use of additive secret sharing to: each secret value x is split into three random shares x_1, x_2, x_3 such that:

$$x_1 + x_2 + x_3 \equiv x \pmod{2^{32}}$$

Sharemind has a well-defined protocol suite that provides support for multiple operations on shared secrets. These include support for comparisons, multiplications, and arithmetic operations in addition to more sophisticated primitives like bit extraction, equality tests, and cross-domain share conversion (from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{32}}$) [39]. These features enable the secure implementation of statistical analysis and machine learning tasks, going beyond simple linear processes and proving the usefulness of additive secret sharing in real applications.

B. Motion Motion [37] is an open-source framework for mixed-protocol N-party MPC, where ”mixed-protocol” refers to the ability to combine several MPC protocols and convert between them so that each protocol’s advantages can be capitalized on. Implemented in C++, Motion is integrated with the HyCC compiler [40] that generates optimized circuits for hybrid MPC protocols, and unlike Sharemind’s fixed three-party model it scales to an arbitrary number of parties and is secure against up to N–1 honest-but-curious adversaries since it implements the Boolean and arithmetic GMW [24] and Beaver–Micali–Rogaway (BMR) [41] (a generalization of Yao’s protocol to multiple parties) as its main backends.

Beyond protocol flexibility, Motion includes various innovative design features that improve performance and modularity. Developers do not need to understand how MPC primitives function with the network stack, instead they can use a convenient provider interface that abstracts primitives like OT, Multiplication Triples [28], Shared Bits, and Square Pairs. Additional features include asynchronous gate evaluation, Single Instruction Multiple Data (SIMD) operations, and efficient conversions between Boolean-GMW, Arithmetic-GMW, and BMR.

Together, these design considerations make Motion a versatile protocol composition platform as well as a scalable, research-oriented MPC framework.

C. Symphony Symphony [38] is an expressive functional domain-specific language for MPC that provides advanced features for coordinating computations with large numbers of parties. The approach is based on Wysteria [42], which introduced static `par` annotations to define which parties should participate in a computation. Symphony greatly expands on this approach by introducing first-class party sets, which are runtime variables rather than static annotations. This allows party sets to be computed, passed, and stored dynamically, allowing for more flexible coordination patterns than Wysteria's compile-time boundaries.

These features enable mechanisms like delegation and resharing, which allow computations or data to be securely transmitted to other groups of parties without reconstruction. The current implementation supports both 2-party Yao (via EMP) and N-party GMW (via Motion), but since the protocol is explicitly chosen in the program text, Symphony is not fully agnostic to the underlying cryptographic framework. Symphony's main contribution is its high-level expressiveness, which enables developers to specify complex, modular MPC programs with dynamic participation and reusable components, supported by a formal calculus (λ-Symphony) with soundness guarantees.

2.6. Choreographies for Secure Computation

While the frameworks mentioned above constitute the state of the art in MPC systems, there is a parallel line of work with the objective to connect secure computation with global coordination models. This section explores how choreographic programming and new compiler-based approaches try to bridge that gap.

In addition to these frameworks, it is essential to explore how choreographic programming and secure multiparty computation are beginning to converge. As previously stated, choreographic languages are primarily concerned with coordination and communication safety, guaranteeing that send/receive pairs are appropriately aligned and that distributed programs are deadlock-free when developed via endpoint projection. However, while expressive, languages like HasChor and MultiChor were not originally designed with secure computation in mind and thus lack native abstractions for cryptographic operations.

For instance, although HasChor offers a clean monadic interface embedded in Haskell, it fails to support multiply-located values or census polymorphism. As a result, implementing

MPC protocols that involve numerous computing parties necessitates duplicating logic for every participant, making it impractical for complex protocols. MultiChor overcomes these limits by including census-polymorphic programming and multiply-located values, resulting in more compact and expressive choreographies. It even has isolated examples of cryptographic protocols in its public repository [19], including the MPC protocol GMW, as well as primitives such as Oblivious Transfer and Diffie–Hellman key exchange. However, their implementations are self-contained, with no reusable abstraction layer or common interface to facilitate further protocol development.

Contrarily, MPC frameworks focus on privacy and cryptographic correctness within formal security models. Since coordination is not their main concern, they hardly ever offer high-level abstractions for global control flow or coordination logic, thus requiring developers to manually control message exchanges and ensure that the intended coordination is respected, i.e., that sends and receives match correctly. Additionally, the round structure is firmly ingrained in the models and security proofs of the majority of MPC protocols, which are made to function in clearly defined rounds of interaction between the participants. As seen previously, Symphony [38] is a well-known general-purpose MPC language that distinguishes itself by explicitly combining safe computing with coordination concepts comparable to choreographic programming. One of these common features is that it offers a scoped parallel execution structure based on generalized SIMD semantics, which enables different parties to use par blocks and first-class party sets to run distinct code paths within the same global structure. Also, similar to choreographic programming languages, these features guarantee that everyone maintains a consistent perspective of logical values, preventing the possibility of mismatched communication or deadlocks.

Additionally, more recent efforts have started to bridge the gap between global coordination models and secure computation. A known solution that focuses on producing high-level, centralized programs with fine-grained security labels to secure distributed code is the Viaduct compiler [43]. Developers in Viaduct tag programs with integrity and confidentiality policies, and the compiler determines the least authority needed for each component, chooses appropriate cryptographic protocols, such as zero-knowledge proofs, commitments, and or MPC, and distributes computation among hosts in compliance with it. Despite not having clearly specified global protocols or endpoint projection, Viaduct's program partitioning secure architecture can be seen as functionally analogous to some aspects of choreographic compilation, even though it does not support choreographic programming in the traditional sense. In particular, it offers a compromise between high-level

security abstractions and the low-level realities of cryptographic enforcement in distributed environments by using security types to guide both control flow and backend selection.

A more recent and directly relevant work that links these two fields is the formal secure synthesis of distributed cryptographic applications [44]. This paper expands on Viaduct's program-partitioning architecture by presenting a formal proof of simulation-based security that holds even in the presence of numerous cryptographic systems, asynchronous communication, and hostile adversaries. To do this, the authors base the compilation process on a security-typed choreography language that represents both high-level centralized programs and low-level distributed protocols at a middle level. This choreographic abstraction allows for global control flow and coordination, as well as endpoint projection into message-passing distributed code that is deadlock-free and security-preserving. Following that, the method integrates this pipeline into the Universal Composability paradigm for secure instantiation of cryptographic primitives and modular reasoning. During the process, it combines the global reasoning discipline of choreographic programming with the backend selection and security type techniques of Viaduct to present a formal basis that unifies both viewpoints into a single end-to-end secure compilation paradigm.

Collectively, these works indicate that although MPC and choreographic programming have evolved mainly separately, there is increasing awareness that their combination can offer robust security and coordination assurances. This creates an active research avenue for combining the assurances of both worlds into a single, cohesive framework.

3. ChoreoMPC: Secure Multiparty Computation as Functional Choreographies

The approach used to investigate the application of choreographic programming for secure multiparty computation is presented in this chapter. We lay out the design and implementation of an embedded domain-specific language in Haskell for secret-sharing-based MPC protocols, building on the foundation established in the previous chapter. The implementation leverages the MultiChor library to orchestrate distributed execution, while aiming to offer high-level abstractions inspired by functional MPC languages such as [38, 45]. We describe the eDSL's structure, talk about how protocols are represented as choreographies, and show a few examples of how to implement MPC protocols.

3.1. Motivation for an eDSL

As discussed previously, choreographic programming languages and secure multiparty computation frameworks each address different aspects of distributed systems design. Choreographies ensure communication correctness and coordination structure, while MPC frameworks focus on cryptographic security.

Symphony demonstrates that such integration is possible, supporting both dynamic coordination and secure computation with multiple cryptographic backends. However, Symphony is not a choreographic language in the formal sense: it lacks support for endpoint projection, and its implementation exists as a standalone DSL with a custom interpreter. As such, it cannot directly leverage the type systems, tooling, or familiarity of established general-purpose languages.

Languages such as Viaduct and its formalization [44] have gone so far as to include cryptographic compilation and security typing, as well as choreographic reasoning in their formal model. Viaduct provides support for numerous cryptographic backends via an extendable runtime, including commitments, zero-knowledge proofs, and two-party MPC via the ABY framework. Its design, however, focuses on secure computation in general, and its MPC support is limited to two parties. In particular, its MPC support is limited to two parties, and its high-level model uses security labels to reason about confidentiality and integrity, which differs from the abstractions commonly seen in choreographic programming. Our goal is orthogonal: to integrate safe choreographic programming as a first-class construct into a host language such as Haskell, building on existing infrastructure such

as MultiChor and exploiting the existing rich type system, functional composition, and abstraction techniques. Unlike Viaduct's closed, tightly integrated design, our eDSL is specialized for general MPC and enables flexible substitution of backends such as Motion and Sharemind within a general-purpose functional environment.

By contrast, MultiChor, a choreographic library embedded in Haskell, provides a rich foundation for writing global specifications with strong safety guarantees. It supports advanced features such as census polymorphism and multiply-located values, which are especially relevant for secure computation involving dynamic sets of parties. Yet, despite its expressiveness, MultiChor lacks built-in abstractions for cryptographic operations or backend integration.

This gap motivates the development of a dedicated eDSL for secure multiparty computation, built directly on top of MultiChor. The proposed eDSL allows protocols to be expressed in a global, choreographic style, while handling cryptographic concerns through a backend-agnostic interface. Implemented entirely in Haskell/MultiChor, it contributes

- A type-class based interface for secret-shared values that integrates cryptographic actions (e.g., secret sharing, resharing, secure arithmetic, reveals) directly with choreographic control flow,
- Backend polymorphism, allowing the same protocol code to run unchanged over multiple MPC frameworks,
- A bridge between MPC and choreographies, demonstrating that MPC programs can be expressed entirely within choreographic abstractions,
- Reuse of Haskell's ecosystem, enabling standard functional constructs (lists, recursion, higher-order functions) to be combined with secure computation in a natural way.

To accomplish this, we developed our own versions of the Sharemind and Motion MPC backends by re-implementing their core primitives and algorithms. This provides a practical and extensible platform for writing secure distributed protocols as composable, choreographed Haskell programs, while giving programmers the flexibility to choose the MPC backend for their computation.

3.2. Implementation Overview

This section describes the concrete implementation of our eDSL. We begin with a review of the basic abstractions that allow for secure, choreographed computing, followed by a practical example: a re-implementation of the Hamming distance protocol originally implemented in Symphony [46], expressed through our own interface.

3.2.1 High-Level Design and API Structure

We will now take a closer look at the eDSL design. This covers a description of the fundamental type classes, data representations, and combinators that allow for systematic protocol creation across several backends.

We begin by defining the central type class `MPC`, shown in Figure 3.1, which specifies the interface that any secure multiparty computation protocol must implement in our system. The class is parameterized by the MPC backend `protocol`, a monadic execution context `m` (typically IO), a set of parties `ps`, and a data type `a`. At the core of this abstraction is the associated type `Secret protocol ps a`, representing a secret-shared value of type `a` held by the coalition `ps`. While the actual implementation of `Secret` is left abstract, it is commonly defined using MultiChor's `Faceted` values, which are well-suited for representing secret shares, as illustrated in Chapter 2.4.3.

It is important to note that not all of these operations preserve the same coalition of parties. Functions such as `classify` and `declassify` are local to the coalition `ps` that participates in the computation, and therefore both their inputs and outputs remain within the same group of parties. By contrast, `share` and `unshare` mediate between different coalitions: they involve the coalition of computing parties `ps`, the group of senders or receivers `qs`, and their union `all`. This distinction explains why their type signatures, as shown in Figure 3.1, explicitly refer to multiple party sets, unlike the operations that remain confined to a single coalition.

To move between public and secret contexts, the `MPC` class provides several key operations: `share`, which distributes a public value as a secret; `unshare`, which reconstructs a secret into a public value; and their optimized local counterparts, `classify` and `declassify`, which avoid communication when the value is already known within the sharing coalition.

In addition, the eDSL provides two distinct operations for refreshing secret shares. The `reshare` operation re-randomizes an existing secret-shared value, producing a fresh sharing of the same underlying secret among the same set of parties. This is particularly useful for preserving security in extended computations, especially when shares are reused or updated. In protocols where the set of computing parties changes dynamically, the more general `reshareTo` operation makes it possible to re-share a value to a potentially different coalition. In our interface, reshareTo has a default implementation in which each participant re-shares their own portion of the secret with the new coalition, and the receivers locally reconstruct their share. All of these operations are orchestrated within MultiChor's `Choreo` monad, ensuring that coordination remains explicit and consistent with choreographic principles.

```
class MPC protocol m ps a where
  data family Secret protocol ps a :: Type
  share     :: Public qs a -> Choreo all m (Secret protocol ps a)
  classify  :: a -> Choreo ps m (Secret protocol ps a)
  unshare   :: Secret protocol ps a -> Choreo all m (Public qs a)
  declassify :: Secret protocol ps a -> Choreo ps m a
  reshare   :: Secret protocol ps a -> Choreo ps m (Secret protocol ps a)
  reshareTo :: Secret protocol ps a -> Choreo all m (Secret protocol qs a)
```

Fig 3.1 - MPC Central type class

Following the core `MPC` interface, we define the type class `SNum`, shown in Figure 3.2, which is a secure version of Haskell's regular `Num` class for arithmetic operations on secret-shared data. This class assumes an `MPC` instance and a concrete `Num` instance for the underlying type `a`. It offers high-level primitives for secure addition, subtraction, negation, and multiplication of secret values, all choreographed within the party set `ps`. The methods `add`, `negate`, and `mul` compute secure addition, negation, and secure multiplication, respectively. The method `minus` has a default implementation as a combination of negation and addition, which is used in case a backend does not provide its own instance. However, specific backends may override this definition with more efficient functions. Since several MPC frameworks include specialized protocols for common operations, keeping the implementation open allows each backend to adopt the most efficient strategy. The class also provides `mulConst`, a version of `mul` that multiplies a secret-shared value by a known constant, implemented by first classifying the constant as a secret value and then using the `mul` function. As with other eDSL methods, all activities are choreographed using the `Choreo` monad to ensure proper coordination and execution among participants.

```
class (MPC protocol m ps a, Num a) => SNum protocol m ps a where
    add :: Secret protocol ps a -> Secret protocol ps a -> Choreo ps m
    (Secret protocol ps a)
    minus :: Secret protocol ps a -> Secret protocol ps a -> Choreo ps m
    (Secret protocol ps a)
    minus x y = negate y >>= add x
    negate :: Secret protocol ps a -> Choreo ps m (Secret protocol ps a)
    mul :: Secret protocol ps a -> Secret protocol ps a -> Choreo ps m
    (Secret protocol ps a)
    mulConst :: Secret protocol ps a -> a -> Choreo ps m (Secret protocol ps
    a)
    mulConst x c = classify c >>= mul x
```

Fig 3.2 - SNum type class

Similarly to the earlier definition of a secure `Num` class, the eDSL includes the `SIntegral` type class, shown in Figure 3.3, which supports integer division over secret-shared variables. It functions as a secure solution for Haskell's conventional `Integral` class. This class introduces secure division primitives and requires an instance of both `MPC` and `Num` for the type `a`. The division of two secret-shared values is done by the `div` method. For the special case of division by a constant, the eDSL provides the method `divConst`, which is implemented by first classifying the constant as a secret value and then performing the usual `div` operation. However, this does not imply that the MPC backend must compute `divConst` in this manner; for example, the Sharemind backend has a specialized protocol for constant division, which we will look at later in the following chapter.

```
class (MPC protocol m ps a, Num a) => SIntegral protocol m ps a where
  div      :: Secret protocol ps a -> Secret protocol ps a -> Choreo ps m
    (Secret protocol ps a)
  divConst :: Secret protocol ps a -> a -> Choreo ps m (Secret protocol ps a)
  divConst x c = classify c >>= div x
```

Fig 3.3 - SIntegral type class

The support for secure boolean logic in our interface is introduced by the definition of the

`SBool` type class, illustrated in Figure 3.4, which defines boolean operations over secret-shared values of type `Bool`. `SBool`, like other eDSL classes, extends the `MPC` instance to provide secure versions of typical boolean operators such as `xor`, `or`, `and`, and `not`. These functions accept secret-shared boolean inputs and return new secret-shared outcomes for the same set of parties. The methods `xor` and `and` securely compute the exclusive-or ($\oplus$) and bitwise-and of two secret values, respectively. The `or` method is defined in terms of `xor` and `and`, following the boolean identity $a \vee b = (a \oplus b) \oplus (a \wedge b)$, which allows it to be expressed as a composition of other secure primitives.

```
class MPC protocol m ps Bool => SBool protocol m ps where
  xor :: Secret protocol ps Bool -> Secret protocol ps Bool -> Choreo ps m
    (Secret protocol ps Bool)
  or  :: Secret protocol ps Bool -> Secret protocol ps Bool -> Choreo ps m
    (Secret protocol ps Bool)
  or i j = xor i j >>= \x -> and i j >>= \y -> xor x y
  not :: Secret protocol ps Bool -> Choreo ps m (Secret protocol ps Bool)
  not x = classify True >>= xor x
  and :: Secret protocol ps Bool -> Secret protocol ps Bool -> Choreo ps m
    (Secret protocol ps Bool)
```

Fig 3.4 - SBool type class

Additionally, the interface provides an `overlappable` instance of `SNum` for `Bool`, shown in Figure 3.5, since boolean arithmetic in $\mathbb{F}_2 \pmod 2$ correlates with logical operations. In this case, `add` and `minus` are implemented as `xor`, while `mul` is implemented as `and`. Because arithmetic negation in $\mathbb{F}_2$ is the identity function, `negate` is implemented as `return`. Multiplication by a constant is optimized by returning either the input or the constant `False`, depending on its value. Marking this instance as `overlappable` makes it a default definition: it is used whenever no more specific instance is available, but concrete backends are free to override it with specialized implementations. This design mirrors the API in general, in that default definitions are offered for convenience while protocol-specific optimizations are allowed when available. These specifications enable `Bool` to be used smoothly with arithmetic interfaces while maintaining the intended boolean semantics.

```haskell
instance {-# OVERLAPPABLE #-} SBool protocol m ps => SNum protocol m ps Bool
    where
    add = xor
    minus = xor
    negate = return
    mul = and
    mulConst x c = if c then return x else classify False
```

Fig 3.5 - Instance of SNum for Bool $(\mathrm{mod}\ 2)$

In MPC, converting integers to their bit-level representations is a common necessity. While arithmetic sharing is useful for operations like addition and multiplication, many essential tasks like comparisons, equality checks, conditional branching, and bitwise operations are better described in the Boolean domain. To suit such use scenarios, MPC systems often include bit decomposition and reconstruction procedures, allowing protocols to switch between arithmetic and Boolean encodings as needed.

Our eDSL handles bit-level manipulation of secret-shared values through the `SBits` type class, shown in Figure 3.6. This class abstracts the ability to convert a secret value of type `a` to its bitwise representation and back. The `SBits` class extends `MPC` with an associated type family `SBitSize a`, which determines, at the type level, how many bits are required to represent values of type `a`. This enables type-safe reasoning about fixed-width bitvectors across different backends. In practice, `SBits` is only instantiated for fixed-size unsigned numeric types, which represent non-negative integers modulo 2^k for `k = SBitSize a`.

This class is made up of two methods: `toBits` and `fromBits`. The `toBits` method receives a secret-shared value and returns a fixed-size vector of secret boolean values wrapped in the `SecretBits` newtype. In contrast, `fromBits` reconstructs the original secret-shared value using its bitwise form. Our eDSL uses a little-endian layout, placing the least significant bit at the head of the vector, following the convention adopted in the DSL by Laud and Randmets [45].

In secure multiparty computation, it is often necessary to operate on values of different bit widths, for instance, promoting a 32-bit value to 64 bits to accommodate larger computations or backend requirements. The `bitExtend` function supports this by padding the most significant bits of a secret-shared value with zeros, thereby extending its size while

preserving its numerical value. This operation is defined generically for any pair of types `a` and `b` where `SBitSize a <= SBitSize b`, allowing bit extension to be expressed in a type-safe and reusable way.

```
type SecretBool protocol ps = Secret protocol ps Bool
type SecretBools protocol ps n = V.Vector n (SecretBool protocol ps)
newtype SecretBits protocol ps a = SecretBits { unSecretBits :: SecretBools
    protocol ps (SBitSize a) }

class (MPC protocol m ps a) => SBits protocol m ps a where
    type SBitSize a :: Nat
    toBits :: Secret protocol ps a -> Choreo ps m (SecretBits protocol ps a)
    fromBits :: (SecretBits protocol ps a) -> Choreo ps m (Secret protocol ps
    a)
```

Fig 3.6 - SBits type class

In order help with conversions between secret-shared values of different kinds, our interface further offers the `SCast` type class, shown in Figure 3.7. This class abstracts the concept of casting a value of type `a` into another type `b`, both securely shared among the same party set. It requires `MPC` instances for both types and includes a single method `cast` offering several default instances for common cases.

When casting a type to itself, the identity function is used. When casting between two bit-compatible types with `SBitSize a <= SBitSize b`, the default instance utilizes `toBits` to convert the source value to its bitwise representation, `bitExtend` to pad the result to the proper width, and `fromBits` to reconstruct the final type. When casting numeric values to boolean values it uses `toBits` and then folds them with the secure `or` function.

```
class (MPC protocol m ps a,MPC protocol m ps b) => SCast protocol m ps a b where
     cast :: Secret protocol ps a -> Choreo ps m (Secret protocol ps b)

instance {-# OVERLAPPABLE #-} (MPC protocol m ps a) => SCast protocol m ps a a where
    cast = return

instance {-# OVERLAPS #-} (MPC protocol m ps Bool,SBitSize a <= SBitSize b,SBits ↩
↪ protocol m ps a,SBits protocol m ps b) => SCast protocol m ps a b where
    cast x = toBits x >>= bitExtend >>= fromBits

instance {-# OVERLAPPABLE #-} (SBits protocol m ps a,SBool protocol m ps) => SCast ↩
↪ protocol m ps a Bool where
    cast x = toBits x >>= \(SecretBits as) -> classify False >>= \f -> V.foldM or f as
```

Fig 3.7 - SCast type class

To compare secret-shared values, our eDSL defines the `SEq` type class, shown in Figure 3.8, which contains a single method, `eq`. This function returns a secret-shared boolean that indicates whether two secret-shared values of the same type are equal. Additionally, for numerical types our eDSL provides a default instance of `eq` that acts as follows: it securely subtracts one value from another using the `minus` method, and then it applies `cast` to turn the result into a boolean. When the two values are equal, their difference becomes zero, therefore the boolean returned by `cast` must be inverted with the not function to match the expected semantics, resulting in a secret-shared `True` when both values are equal.

```
class SEq protocol m ps a where
    eq :: Secret protocol ps a -> Secret protocol ps a -> Choreo ps m (Secret
    protocol ps Bool)

instance {-# OVERLAPPABLE #-} (SNum protocol m ps a,SCast protocol m ps a
    Bool, SBool protocol m ps) => SEq protocol m ps a where
    eq x y = do
        d <- minus x y
        r <- mul d d -- this is 0 iff x=y, otherwise some nonzero value
        cast r >>= not
```

Fig 3.8 - SEq type class

To handle conditional statements securely, our interface defines the `SMux` type class, shown in Figure 3.9, which includes a single method named `mux`. The name is derived from the hardware concept of a multiplexer, which is a component that selects one of

several inputs based on a control signal. Conceptually, `mux` mirrors the conditional multiplexer construct of Symphony, where both branches are evaluated, and the final result is selected securely, without branching on public control flow. That is, if the secret predicate `b` is `True`, `mux b x y` yields `x`; if not, it yields `y`. For numerical types, the default instance of `mux` implements this selection using the conventional masked formula $b * x + (1 - b) * y$, where `b` is interpreted as either 0 or 1 in the numeric domain. This works because if `b = 1`, the outcome is $x + 0 * y = x$, whereas if `b = 0`, the result is $0 * x + 1 * y = y$. Since both branches are computed and combined arithmetically, the selection action avoids public branching while maintaining the secrecy of `b`.

```
class SMux protocol m ps a where
    mux :: Secret protocol ps Bool -> Secret protocol ps a -> Secret protocol
    ps a -> Choreo ps m (Secret protocol ps a)

instance {-# OVERLAPPABLE #-} (SNum protocol m ps a,SCast protocol m ps Bool
    a) => SMux protocol m ps a where
    mux = muxDefault

muxDefault :: (SCast protocol m ps Bool a,SNum protocol m ps a) => Secret
    protocol ps Bool -> Secret protocol ps a -> Secret protocol ps a ->
    Choreo ps m (Secret protocol ps a)
muxDefault bool x y = do
    bool' <- cast bool
    bx <- mul bool' x
    by <- classify 1 >>= \one -> minus one bool' >>= \b1 -> mul b1 y
    add bx by
```

Fig 3.9 - SMux type class

To support secure ordering comparisons, our eDSL specifies the `SOrd` type class, illustrated in Figure 3.10, which includes four methods: `lt` (less than), `le` (less than or equal), `gt` (greater than), and `ge` (greater than or equal). Every method returns a secret-shared boolean that reflects the outcome of the comparison between two secret-shared values of the same type. Our default instance uses the operations from the `SBool` type class to implement these methods in the boolean domain. To calculate `lt a b`, for instance,

`not` is applied to `a` first, and the result is then safely `and`-ed with `b`. The remaining methods are defined in an analogous manner.

For general numeric types, the interface offers a bitwise comparison approach. The values are first transformed to their bit representations using `toBits`, then the resulting vectors are reversed so that comparison runs from most significant bit to least significant bit, as in conventional numeric ordering. After that, a fold is then applied across the two inputs' paired bits, executing an algorithm an algorithm that checks if the current bit position shows that one value is smaller or larger than the other at every step, and it only changes the result if all of the more significant bits that have been observed thus far are equal. To ensure that no information about intermediate results is disclosed, this is implemented completely using secure boolean operations instantiated earlier.

As with other type classes in our eDSL, MPC backends are free to override these default definitions with more efficient, protocol-specific implementations.

```
class SOrd protocol m ps a where
    lt :: Secret protocol ps a -> Secret protocol ps a -> Choreo ps m (Secret protocol ps Bool)
    le :: Secret protocol ps a -> Secret protocol ps a -> Choreo ps m (Secret protocol ps Bool)
    gt :: Secret protocol ps a -> Secret protocol ps a -> Choreo ps m (Secret protocol ps Bool)
    ge :: Secret protocol ps a -> Secret protocol ps a -> Choreo ps m (Secret protocol ps Bool)

instance {-# OVERLAPPABLE #-} SBool protocol m ps => SOrd protocol m ps Bool where
    lt x y = not x >>= and y
    le x y = not x >>= or y
    gt x y = not y >>= and x
    ge x y = not y >>= or x

instance {-# OVERLAPPABLE #-} (SBits protocol m ps a,SBool protocol m ps) => SOrd protocol m ps a where
    lt x y = do
        SecretBits bx <- toBits x
        SecretBits by <- toBits y
        let go acc (x,y) = do
                x_lt_y <- lt x y
                not_x_lt_y <- not x_lt_y
                x_gt_y <- gt x y
                not_x_gt_y <- not x_gt_y
                mul not_x_lt_y not_x_gt_y >>= mul acc >>= xor x_lt_y
        classify False >>= \false -> V.foldM go false (V.zip bx by)
    gt x y = do
        SecretBits bx <- toBits x
        SecretBits by <- toBits y
        let go acc (x,y) = do
                x_gt_y <- gt x y
                not_x_gt_y <- not x_gt_y
                x_lt_y <- lt x y
                not_x_lt_y <- not x_lt_y
                mul not_x_gt_y not_x_lt_y >>= mul acc >>= xor x_gt_y
        classify False >>= \false -> V.foldM go false (V.zip bx by)
    ge x y = lt x y >>= not
    le x y = gt x y >>= not
```

Fig 3.10 - SOrd type class

Finally, to simplify working with grouped values like tuples and lists in secure multiparty computation, our language defines several helper type classes for handling them in both the secret and party-local domains, illustrated in Figure 3.11. Here, party-local refers to values of type `Public ps a = Located ps a`, meaning the value is known to all parties in the set `ps`, but not necessarily to others in the computation.

- `STuple` abstracts operations on secret-shared tuples, providing methods `sFst` and `sSnd` to securely extract the first or second component, and `sZip` to combine two secret values into a secret tuple.
- `PTuple` offers the same functionality for party-local tuples, with methods `pFst`, `pSnd`, and `pZip` combining two Public/Located values into a Public tuple.

- `SList` supports conversions between a secret-shared list value and a standard Haskell list of secret-shared elements (`secrets2List` and `list2Secrets`), enabling easier iteration, mapping, and folding over secret data. It also includes `unestPublic`, which lifts a public list of secret values into a single secret-shared list, a functionality needed for expressing certain higher-level protocols.
- `PList` mirrors this behavior for party-local lists, providing `pub2List` and `list2Pub` for converting between Haskell lists and Public/Located list values.

```
class SList protocol m ps a where
  secrets2List :: Secret protocol ps [a] -> Int -> Choreo all m [Secret protocol ps a]

  list2Secrets :: [Secret protocol ps a] -> Choreo all m (Secret protocol ps [a])

  unestPublic :: Public ps [Secret protocol ps a] -> Subset ps all -> Choreo all m ↩
↪ (Secret protocol ps [a])

class STuple protocol m ps a b where
  sFst :: Secret protocol ps (a, b) -> Choreo all m (Secret protocol ps a)
  sSnd :: Secret protocol ps (a, b) -> Choreo all m (Secret protocol ps b)
  sZip :: Secret protocol ps a -> Secret protocol ps b -> Choreo all m (Secret ↩
↪ protocol ps (a, b))

class PList ps a where
  pub2List :: Public ps [a] -> Int -> Choreo all m [Public ps a]

  list2Pub :: [Public ps a] -> Choreo all m (Public ps [a])

class PTuple ps a b where
  pFst :: Public ps (a, b) -> Choreo all m (Public ps a)
  pSnd :: Public ps (a, b) -> Choreo all m (Public ps b)
  pZip :: Public ps a -> Public ps b -> Choreo all m (Public ps (a, b))

class (MPC protocol m ps a) => SDefault protocol m ps a where
  sDefault :: Choreo ps m (Secret protocol ps a)

class PDefault m ps a where
  pDefault :: Choreo ps m (Public ps a)
```

Fig 3.11 - Type classes for Lists and Tuples

In addition, our interface defines the `SDefault` and `PDefault` classes, which provide default secret and public values for padding, respectively. Haskell lists such as `[Secret protocol ps a]` or `[Public ps a]` expose their length publicly, whereas arrays of type `Secret protocol ps [a]` and `Public ps [a]` have private sizes. Consequently, the conversion methods `secrets2List` and `pub2List` require an explicit integer size argument that is not necessarily the actual size of the secret array, in order to preserve its secrecy. If the requested size is smaller than the array's true length, the result is truncated; if it is larger, the list is padded with default values obtained from `sDefault` or `pDefault`, respectively. This design ensures

both type safety and correct handling of size mismatches when converting between secret and public list representations.

The eDSL also provides helper functions for classifying, declassifying, sharing, and unsharing tuples, lists and list of tuples, shown in Figure 3.12. When working with structured data, these features minimize the need to write the same setup code over and over again, resulting in shorter protocol code as well as more secure and consistent conversions and extractions.

```
classifyTuple (x,y) = do
    sx <- classify x
    sy <- classify y
    sZip @protocol @m @ps @a @b sx sy

shareTuple pub = do
  sx <- pFst pub >>= share
  sy <- pSnd pub >>= share
  sZip @protocol @m @ps sx sy

declassifyTuple sec = do
    sx <- sFst @protocol @m @ps @a @b sec >>= declassify
    sy <- sSnd @protocol @m @ps @a @b sec >>= declassify
    return (sx, sy)

unshareTuple sec = do
  x <- sFst @protocol @m @ps @a @b sec >>= unshare
  y <- sSnd @protocol @m @ps @a @b sec >>= unshare
  pZip x y

classifyList = mapM classify

declassifyList = mapM declassify

shareList v size = do
     list <- pub2List v size -- Public qs [a] -> [Public qs a] -- defined in PList
     mapM share list

unshareList secrets = do
  let sub = explicitSubset :: Subset qs all
  values <- mapM unshare secrets
  list2Pub values -- [Public qs a] -> Public qs [a] defined in PList

shareTupleList pub size = do
  list <- pub2List pub size  -- -- Public qs [(a, b)] -> [Public qs (a, b)] defined ↩
↪ in PList
  forM list $ \rk -> do
    sx <- pFst rk >>= share
    sy <- pSnd rk >>= share
    sZip @protocol @m @ps sx sy

unshareTupleList list = do
  xs <- forM list $ \s -> do
    x <- sFst @protocol @m @ps @a @b s >>= unshare
    y <-  sSnd @protocol @m @ps @a @b s >>= unshare
    pZip x y
  list2Pub xs -- [Public qs (a, b)] -> Public qs [(a, b)] defined in
```

Fig 3.12 - Helpers for (un)sharing and (de)classifying

3.2.2 A Complete Example: Hamming Distance

To demonstrate the practical use of our eDSL, we now move on to a specific case. We use the Hamming distance, a straightforward yet representative secure computation, as an example. After a quick review of its description and desired calculation, we look at how the protocol can be written in Symphony. Lastly, we demonstrate how our eDSL may be used to create the same protocol, emphasizing the expressiveness that comes from embedding within Haskell and the correspondence between constructs.

The Hamming distance algorithm is a basic tool for calculating how different two pieces of data, usually strings or integers, are from one another. It calculates the number of positions at which the corresponding elements differ, as illustrated in Figure 3.13.

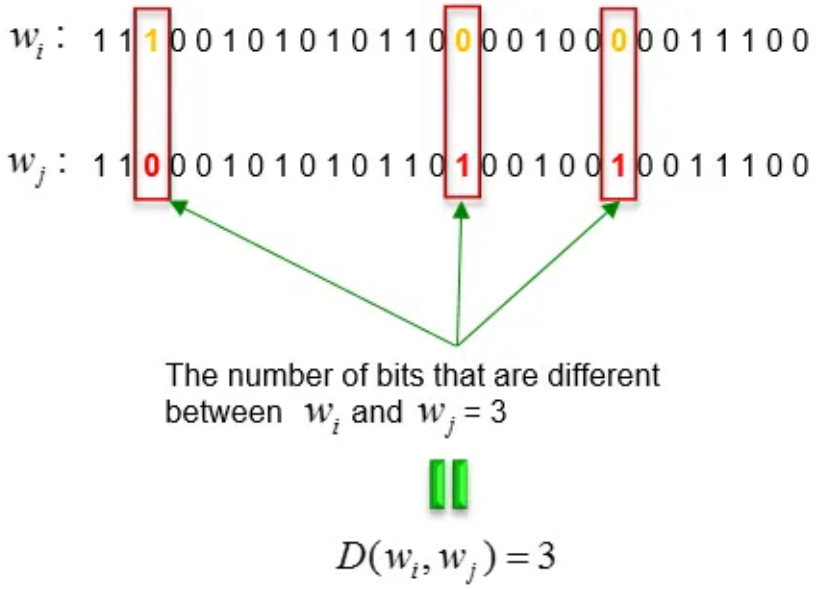

Fig 3.13 - Hamming distance between two strings [47]

We now discuss how the Hamming distance protocol is implemented in Symphony, as shown in Figure 3.14. The program begins with the identification of the participating parties, (A) and (B), using the `principal A B` statement.

The secure computation is defined as a function that takes two arrays of equal, publicly known length. Within this definition, a recursive process iterates over the array indices while maintaining an accumulator for the Hamming distance. At the beginning of each iteration, it publicly checks whether the index has reached the array's length; if so, the accumulated distance is returned. Otherwise, the current elements of the two arrays are compared securely. A multiplexer (`mux`) is then used to convert the comparison result

into an increment: 0 if the elements are equal, or 1 if they differ. This increment is added to the running total, and the recursion proceeds with the next index.

The main function orchestrates the overall computation. To indicate the parties that are active participants, it begins with a `par {A,B}` block, meaning that this computation involves both parties A and B. Each party then reads its own input file in a separate block: Party A loads input A, and Party B loads input B. The inputs are then shared under a backend chosen at the language level, in this case the `gmw` backend. This annotation specifies that the secure computation on these values will be executed according to the logic of the selected backend. Once the inputs are securely shared, the Hamming distance computation is applied to the secret-shared arrays, and the final result is revealed only to Party A.

```
1  principal A B
2
3  def hammingDist bs0 bs1 =
4    let len = size bs0 in
5    let hammingDistRec = fun [hammingDistRec] i distance →
6      if i == len then
7        distance
8      else
9        let inc = mux if bs0.i == bs1.i then 0n else 1n in
10       hammingDistRec (i + 1n) (distance + inc)
11   in hammingDistRec 0n 0n
12
13 def main () = par {A,B}
14   let inputA = par {A} read (array N) from "hamming-1k.txt" in
15   let inputB = par {B} read (array N) from "hamming-1k.txt" in
16
17   let shareA = share [gmw, array N : {A} → {A,B}] inputA in
18   let shareB = share [gmw, array N : {B} → {A,B}] inputB in
19
20   reveal [gmw, N : {A,B} → {A}] (hammingDist shareA shareB)
```

Fig 3.14 - Hamming Distance in Symphony

In our eDSL, the function for computing the hamming distance has the same conceptual structure as the Symphony version, but it is expressed using our type classes' high-level abstractions, such as `SEq`, `SMux`, and `SNum`, as illustrated in Figure 3.15. One key distinction is that our eDSL is integrated into Haskell, while Symphony is a stand-alone DSL with its own interpreter and control systems. We can directly utilize the host language's features thanks to this embedding. For instance, our version's recursive helper does not require a unique looping construct because it is written using standard Haskell recursion. It is possible to use standard functional abstractions without having to re-implement them in our language.

Another important difference concerns how array lengths are treated. In Symphony, programs such as the hamming distance algorithm rely on the input size as an explicit public parameter, so the length of input lists is always public. In contrast, our eDSL distinguishes two types of array representations: values of type `Secret protocol ps [a]`, which have a private length, and lists of type `[Secret protocol ps a]`, which have a public length. In practice, to take advantage of Haskell's recursion, our Hamming distance function operates on the public-length representation.

In addition, party sets are processed differently. In Symphony, party sets are first-class runtime values that can be built, assembled, and annotated in `par` blocks for dynamic coordination. In contrast, our eDSL structures party sets at the type level, making them first-class types. Their structure and membership are statically tracked, and Haskell's type checker requires coordination guarantees prior to execution.

These linguistic differences surface immediately in the implementation. Because the eDSL is embedded, the inputs are conventional Haskell lists with secret values and list processing uses standard recursion. Concretely, our program takes two lists of secret-shared values, computes their (public) lengths, then iterates over the shorter list. The constants `eqValue` (0) and `notEqValue` (1) are classified for future use.

Following this, and similarly to Symphony, a recursive helper is defined that goes through the lists element by element, following these steps:

1. If the index has reached the length of the shorter list, return the accumulated distance.
2. Otherwise, securely compare the current elements at the same position using `eq`.
3. After the comparison, feed the boolean result into `mux` to select either `eqValue` (0) if the elements are equal or `notEqValue` (1) if they differ.
4. Securely add this increment to the running total.
5. Increment iterator index and jump to step 1.

After evaluating the overlapping portion of the lists, the function determines whether one list is longer than another. If so, it securely adds the difference in lengths to account for the additional elements before returning the total secret value.

```
hammingDist :: forall protocol m ps a.
    (MPC protocol m ps a, SEq protocol m ps a,
     SMux protocol m ps a, SNum protocol m ps a) =>
    [Secret protocol ps a] -> [Secret protocol ps a] ->
    Choreo ps m (Secret protocol ps a)
hammingDist bs0 bs1 = do
    let mi = min (length bs0) (length bs1)
        ma = max (length bs0) (length bs1)
    eqValue    <- classify (0 :: a)
    notEqValue <- classify (1 :: a)
    let go i d | i == mi  = return d
               | otherwise = do
                   cond' <- eq (bs0 !! i) (bs1 !! i)
                   inc   <- mux cond' eqValue notEqValue
                   add inc d >>= go (i + 1)
    out <- go 0 eqValue
    if ma /= mi
       then classify (fromIntegral (ma - mi) :: a) >>= add out
       else return out
```

Fig 3.15 - Hamming Distance in our eDSL

Similar to the main function in Symphony's example, we have defined a function that demonstrates how the protocol is orchestrated in practice, as shown in Figure 3.16. Computing party 1 and Computing party 2 each begin with a private input list known only to themselves. We then pass `shareList` a fixed public list size (`publicSize`) that is known to all parties. Internally, `shareList` use `pub2List` to transform `Public ps [a]` into `[Public ps a]` without disclosing the actual length. Consequently, the lists are secret-shared element by element using a `mapM` with our interface's `share` function.

Unlike Symphony, which uses explicit annotations like `principal A B` and `par A,B`, our eDSL relies on Haskell's monadic structure. Party sets are represented at the type level, with `ps` in `Choreo ps m` indicating the coalition of parties involved, so coordination is enforced by the host language's type checker. This makes orchestration follow the style of ordinary Haskell programming while ensuring statically that only the intended parties take part in each phase. As a result, the secret value that is produced is safely kept private and revealed only to the specified output parties.

```haskell
testHamming :: Choreo SharemindCparties IO ()
testHamming = do
    let inputs1 :: Public '["cparty1"] [Word32] = wrap [1,2,3,4]
        inputs2 :: Public '["cparty2"] [Word32] = wrap [1,2,3,4,5]
        publicSize = 10

    aSecrets <- shareList @'["cparty1"] @Sharemind inputs1 publicSize
    bSecrets <- shareList @'["cparty2"] @Sharemind inputs2 publicSize

    dist <- hammingDist aSecrets bSecrets
    u :: Public '["cparty1"] Word32 <- unshare dist
    return ()
```

Fig 3.16 - Main of Hamming Distance in our eDSL

Overall, this example shows a direct correspondence between the Symphony implementation and our eDSL version: they have the same logical structure, but our Haskell embedding allows us to access the host language's strengths directly. Beyond customized abstractions like `SEq`, `SMux`, and `SNum`, we can take advantage of Haskell's ecosystem: predefined list operations, higher-order functions like `map` and `fold`, as well as abstractions provided by functors and monads. In contrast, because Symphony is a stand-alone DSL, it needed a specific standard library to offer such features, including unique implementations of maps, bundles, recursion, and coordination patterns. Furthermore, existing libraries, such as MultiChor and its combinators, can be effortlessly incorporated, enabling secure computations to be expressed with the same tools as ordinary functional programs. As a result, our definition of hamming distance is concise, type-safe, and fits naturally within the broader ecosystem of functional programming. Furthermore, our eDSL generalizes the protocol to inputs of different lengths while maintaining readability and security, demonstrating how functional programming and choreographic principles may be combined to produce protocols that are both expressive and practical.

3.3. MPC Backend Instantiations

In the last section, we introduced our eDSL's high-level API and structure and demonstrated how protocols could potentially be constructed in a backend-independent manner. We now turn to the concrete implementation of this interface by developing two MPC backends within our system: Sharemind and Motion (Section 2.5.4).

As an example of application-level use, we previously showed how secure programs such as the Hamming distance algorithm can be implemented on top of MPC using our eDSL. Here, we show that the protocols that enable MPC itself, like secure multiplication and additive secret sharing, may also be expressed directly in the choreographic dialect. In our

context, these building blocks are re-implemented as MultiChor programs, although they frequently stay hidden behind MPC engines such as Sharemind or Motion. This shows that the same abstractions used for high-level applications are also sufficient to capture the low-level cryptographic techniques used by MPC. The next two sections illustrate how the abstract operations provided in the MPC type classes are implemented for each backend, revealing low-level design choices and protocol-specific constraints.

For Sharemind, this means making use of the specific features and optimizations made for this fixed environment, which offer passive security against a single compromised party (honest-majority security), as well as supporting its native three-party model. Motion, on the other hand, offers passive security against up to $n - 1$ corruptions out of n and can be instantiated with an arbitrary number of computing parties. This variant of the adversarial model highlights the various underlying assumptions of each backend, which are reflected in the implementations of each that are discussed in the following subsections.

Both backend implementations are defined generically over unsigned integer types, that is, fixed-width unsigned integers of size k bits. Semantically, a value of type `Word_k` represents an element of the ring $\mathbb{Z}_{2^k}$, so all arithmetic is performed modulo 2^k. This choice reflects the conventions of frameworks such as Sharemind, which operate over modular arithmetic domains, and allows our implementations to reuse the same code across different bit widths, such as 32-bit or 64-bit words, without modification. It also provides a uniform basis for instantiating both arithmetic circuits (over $\mathbb{Z}_{2^k}$) and Boolean circuits (as the special case $k = 1$, i.e., the field $\mathbb{F}_2$).

Notation To simplify the presentation, we introduce a uniform arithmetic notation that applies to both arithmetic and Boolean settings. We write

$$+', \ -', \ *', \ 0', \ 1'$$

for addition, subtraction, multiplication, and the additive and multiplicative units, respectively. For word types Word_k (i.e., arithmetic over $\mathbb{Z}_{2^k}$), these symbols correspond to standard modular operations and constants. For the Boolean domain ($k = 1$, i.e., $\mathbb{F}_2$), they specialize to field operations: $+'$ and $-'$ denote $\oplus$, $*'$ denotes $\wedge$, and $0', 1'$ correspond to `False` and `True`.

3.3.1 Instantiation of the Sharemind Backend

The instantiation of our eDSL for the Sharemind MPC backend is the main objective of this section. We discuss how each operation in our interface is implemented in Sharemind, while also emphasizing backend-specific aspects such as unique protocols and optimizations that differ from conventional methods.

As previously stated, Sharemind is based on additive secret sharing across a ring of 32-bit unsigned integers, and it is distinguished by a fixed three-party computation architecture in which every secure computation is carried out by exactly three computing parties. A secret value x is represented in this context as three shares, x_1, x_2, and x_3, one held by each of the computing parties $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$, such that:

$$x_1 + x_2 + x_3 \equiv x \pmod{2^{32}}.$$

With this technique, the sharing process is simple: the input party, who is the party who holds the secret, creates two random 32-bit numbers and calculates the third as:

$$third \equiv x - first - second \pmod{2^{32}},$$

then sends the first share to $\mathcal{P}_1$, the second to $\mathcal{P}_2$, and the third to $\mathcal{P}_3$. This guarantees that no single party uncovers anything about x alone, but by adding up all three shares modulo 2^{32}, the initial value can be recovered.

In our Sharemind backend logic, secret sharing is implemented by the `secretShare` function whose type signature is shown in Figure 3.17.

```
1 secretShare ::
2   forall allparties cparties p m a. Random a =>
3   Member p allparties ->
4   Subset cparties allparties ->
5   Located '[p] a ->
6   Choreo allparties m (Faceted cparties '[] a)
```

Fig 3.17 - secretShare type signature

Several of the constraints come from MultiChor's type-level API. More precisely, `KnownSymbol p` and `KnownSymbols cparties` guarantee that the party identifiers mentioned

in the type can be recovered at compile time: their names and the structure of the list are available to the compiler, which allows MultiChor to generate code that is statically aware of the census of parties. The actual subset of computing parties, however, is not fixed at compile time but provided at run time via the `Subset` argument.

The first argument, `Member p allparties`, is a type-level proof constraints that ensures `p` is part of the global set of parties `allparties`, and also serves as a term-level handle to that party in local calculations.

The second argument, `Subset cparties allparties`, is likewise a type-level proof that the set of computing parties `cparties` is contained in `allparties`. As discussed in Section 2.4.3, these proof objects are implemented using the Ghosts of Departed Proofs technique, which enforces correctness at the type level while erasing the proofs at runtime. This makes it possible to restrict communication and computation to a smaller coalition, while still keeping the overall choreography monad indexed by the full set `allparties`.

The final argument is the secret to be shared, represented as a `Located` value located at party `p`.

The body of `secretShare` works as follows: We call `locally p` with an auxiliary function to generate the additive secret shares for the computing parties as explained before, yielding a `Located '[p] (Quire cparties a)`. The shares are then distributed using `scatter`, which sends each element of the `Quire` to its corresponding party, followed by `othersForget` to strip out the sender information. All of these primitives are provided by the MultiChor API (Section 2.4.3). With this, we obtain a `Faceted` value containing the additive shares at `cparties`, ready to be used in secure computation.

Once the computation is complete, the result must be provided to a group of parties known as the output parties, and as previously explained, revealing an additive share is as simple as adding all of the shares. In order to accomplish this, we define the three-argument function `unShare`, which type signature is shown in Figure 3.18. The first two arguments are subset proofs, one for the output parties and one for the computing parties. Finally, it also takes the faceted shares over the computing parties and produces the revealed value, represented as a `Located outputparty a`.

```haskell
unShare ::
  forall cparties allparties outputparty m a.
  ( KnownSymbols cparties
  , KnownSymbols outputparty
  , MonadIO m, Show a, Read a
  ) =>
  Subset cparties allparties ->
  Subset outputparty allparties ->
  Faceted cparties '[] a ->
  Choreo allparties m (Located outputparty a)
```

Fig 3.18 - unShare type signature

The definition of `unShare` relies on the MultiChor primitives `gather` and `congruently` (Section 2.4.3). Intuitively, `gather` collects the shares from the computing parties and delivers them to the output parties, and `congruently` has all output parties locally combine the shares by addition to reconstruct the secret.

The functions for classifying and declassifying in Sharemind's fixed three-party model are straightforward. To classify a public value x as an additive sharing among the three computing parties ($\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$), we can deterministically create the share triple $(x, -x, x)$ over the underlying ring, which eliminates the need for randomness while ensuring that the shares add up to the initial value.

Declassification is also straightforward: $\mathcal{P}_1$ receives the shares from parties $\mathcal{P}_2$ and $\mathcal{P}_3$, reconstructs the secret by locally adding up all three shares and the result is then broadcast to all parties in the computation, resulting in a `Located cparties a`. In the end, this located value that is known by the entire census is unwraped by MultiChor's `naked` function, making it accessible to all intended recipients in plain form.

A fundamental component of several of Sharemind's protocols is the `reshare` primitive. Resharing is crucial to maintaining share independence in Sharemind's three-party computation model given that newly created shares with the same secret are statistically independent from the ones before them, avoiding any potential leakage that would result from reusing shares over several processes. This is especially crucial when an intermediate value will be used as input for subsequent protocols since, in the absence of re-randomization, the share structure may disclose incomplete information over time.

Our implementation of `reshare` in the Sharemind backend follows the approach described by Bogdanov et al.[39] and Laud et al.[45], reproduced in Figure 3.19.

Algorithm 1: Resharing protocol $[\![w]\!] \leftarrow \text{Reshare}([\![u]\!])$.

Data: Shared value $[\![u]\!]$.
Result: Shared value $[\![w]\!]$ such that $w = u$, all shares w_i are uniformly distributed and u_i and w_j are independent for $i, j = 1, 2, 3$.

1 $\mathcal{P}_1$ generates random $r_{12} \leftarrow \mathbb{Z}_{2^n}$.
2 $\mathcal{P}_2$ generates random $r_{23} \leftarrow \mathbb{Z}_{2^n}$.
3 $\mathcal{P}_3$ generates random $r_{31} \leftarrow \mathbb{Z}_{2^n}$.
4 All values $*_{ij}$ are sent from $\mathcal{P}_i$ to $\mathcal{P}_j$.
5 $\mathcal{P}_1$ computes $w_1 \leftarrow u_1 + r_{12} - r_{31}$.
6 $\mathcal{P}_2$ computes $w_2 \leftarrow u_2 + r_{23} - r_{12}$.
7 $\mathcal{P}_3$ computes $w_3 \leftarrow u_3 + r_{31} - r_{23}$.
8 Return $[\![w]\!]$.

Fig 3.19 - Reshare pseudocode [39]

In our implementation, all computations are carried out locally, while communication between the computing parties is handled via HasChor/MultiChor's default send operator `~>`. Finally, to reconstruct faceted values from the three individual party-local values, we built a helper function `locs2Faceted`, as shown in Figure 3.20.

```
locs2Faceted
  :: Located '[p1] a
  -> Located '[p2] a
  -> Located '[p3] a
  -> Faceted '[p1, p2, p3] '[] a
locs2Faceted la lb lc = PIndexed $ \case
  First             -> Facet la
  Later First       -> Facet lb
  Later (Later First) -> Facet lc
```

Fig 3.20 - Helper: Faceted value from three Located

In addition to the conventional `reshare` primitive, Sharemind provides the reshareToTwo protocol, which also produces new statistically independent shares of a secret, but with the particularity that one of the three computing parties receives a zero share, hence the name "reshare to two". This operation is often used in optimizations or subprotocols that aim to

reduce communication and computation costs when the computation can be restricted to a subset of the parties.

Figure 3.21 shows how our Sharemind backend uses the Bogdanov et al.[39] and Laud et al.[45] approach for the reshareToTwo protocol.

Algorithm 10: Protocol $[\![u']\!] \leftarrow$ ReshareToTwo($[\![u]\!]$) for resharing a value $[\![u]\!]$ between the parties $\mathcal{P}_2$ and $\mathcal{P}_3$.

Data: Shared value $[\![u]\!]$.
Result: Shared value $[\![u']\!]$ so that $u = u'$ and $u'_1 = 0$.
1 $\mathcal{P}_1$ generates random $r_2 \leftarrow \mathbb{Z}_{2^n}$ and computes $r_3 \leftarrow u_1 - r_2$.
2 $\mathcal{P}_1$ sets $u'_1 = 0$ and sends r_i to $\mathcal{P}_i$ $(i = 2, 3)$.
3 $\mathcal{P}_i$ computes $u'_i \leftarrow u_i + r_i$ $(i = 2, 3)$.
4 Return $[\![u']\!]$.

Fig 3.21 - ReshareToTwo pseudocode [39]

In contrast, the more general `reshareTo` operation does not have a dedicated protocol in Sharemind's fixed three-party model. For this reason, our implementation simply relies on the default generic definition of `reshareTo` provided by the MPC interface.

After discussing the essential data management primitives that allow values to be distributed, retrieved, and re-randomized securely, we proceed on to the basic operations that support the majority of MPC protocols. In cryptography, computation is typically represented as circuits, with the functionality to be computed expressed in terms of gates over an underlying ring or field. In this case, secure addition and secure multiplication are sufficient basic building blocks because any arithmetic circuit can be described by these two operations. Following the definition of these primitives in [35, 39], we examine Sharemind-specific primitives for bit shifts, equality and comparison (GTE) evaluations, bitwise and arithmetic share conversion, and public and private division support.

Core Arithmetic Primitives As was previously said in relation to circuit-based, multi-party protocols, MPC based on linear secret sharing schemes enables some tasks to be performed locally, without requiring any communication between the parties. Addition fits under this category in Sharemind's additive sharing structure, where each party only adds its share of the two inputs to determine the sum of two secret-shared values. Subtraction operates similarly, with each side subtracting its share. Finally, negation can be computed locally as well, by having each party replace its share x_n with $0' - ' x_n$. Additionally, when the multiplier is public, multiplication admits a local case: each party merely scales

its share by that constant, resulting in the product without any interaction.

$$\text{i.e., } x \cdot a \equiv \sum_{i=1}^{3} x_i \cdot a$$

Multiplication between two secret shared values, on the other hand, needs interaction between the computing parties. Our implementation follows the approach in Figure 3.22 from [39], where both operands are initially re-randomized using the `reshare` primitive to ensure the shares are statistically independent. Following that, the calculation relies on the distributive property of multiplication over addition, for instance considering $u, v \in \mathbb{Z}_{2^{32}}$ two additively shared values:

$$(u_1 + u_2 + u_3)(v_1 + v_2 + v_3) = \sum_{i=1}^{3}\sum_{j=1}^{3} u_i v_j$$

Which can be rewritten as:

$$\sum_{i=1}^{3}(u_i v_i + u_i v_{p(i)} + u_{p(i)} v_i),$$

where $p(i)$ denotes the "previous" party in the fixed three-party order $(p(1) = 3, p(2) = 1, p(3) = 2)$. This approach allows each party to compute the three terms locally using only its own shares (u_i, v_i) and those of its previous neighbor $(u_{p(i)}, v_{p(i)})$, after which the partial results are summed and the product is reshared to yield a fresh, secure sharing of the result.

Algorithm 2: Protocol for multiplying two shared values $[\![w']\!] \leftarrow \mathsf{Mult}([\![u]\!], [\![v]\!])$.

Data: Shared values $[\![u]\!]$ and $[\![v]\!]$.
Result: Shared value $[\![w']\!]$ such that $w' = uv$.

1 $[\![u']\!] \leftarrow \mathsf{Reshare}([\![u]\!])$
2 $[\![v']\!] \leftarrow \mathsf{Reshare}([\![v]\!])$
3 $\mathcal{P}_1$ sends u'_1 and v'_1 to $\mathcal{P}_2$.
4 $\mathcal{P}_2$ sends u'_2 and v'_2 to $\mathcal{P}_3$.
5 $\mathcal{P}_3$ sends u'_3 and v'_3 to $\mathcal{P}_1$.
6 $\mathcal{P}_1$ computes $w_1 \leftarrow u'_1 v'_1 + u'_1 v'_3 + u'_3 v'_1$.
7 $\mathcal{P}_2$ computes $w_2 \leftarrow u'_2 v'_2 + u'_2 v'_1 + u'_1 v'_2$.
8 $\mathcal{P}_3$ computes $w_3 \leftarrow u'_3 v'_3 + u'_3 v'_2 + u'_2 v'_3$.
9 Return $[\![w']\!] \leftarrow \mathsf{Reshare}([\![w]\!])$.

Fig 3.22 - Multiplication pseudocode [39]

In our implementation, both gates are defined using MultiChor's `parallel` combinator, which executes the same local computation at each computing party and aggregates the results into a `Faceted` value. Both addition and multiplication gates, as well as the reshare

function used in multiplication, are implemented generically over word types, i.e., values in $\mathbb{Z}_{2^k}$. To keep the definitions uniform across settings, we rely on the abstract operators introduced in our notation 3.3. In the arithmetic case ($k > 1$), these correspond to standard modular $+$, $-$, and $*$, while in the Boolean case ($k = 1$) they specialize to $\oplus$ and $\wedge$. As a result, Boolean shares can be handled by the same addition and multiplication gate definitions, which then behave as XOR and AND gates in the field $\mathbb{F}_2$.

Share Conversion Functions For converting between different secret-sharing domains, Sharemind offers two fundamental primitives: ShareConv and BitExtraction. The ShareConv operation converts a secret value from the field $\mathbb{F}_2$ (a single secret-shared Boolean) into $\mathbb{F}_{2^{32}}$ (an arithmetic sharing over 32-bit), while BitExtraction performs the other way around, decomposing a value in $\mathbb{F}_{2^{32}}$ into a vector of secret-shared bits representing its binary representation.

Our ShareConv implementation is based on the algorithm that is presented in Figure 3.23 and explained in Bogdanov et al. [39]. In essence, the protocol randomizes the input Boolean shares, performs a local conversion of one party's share into the arithmetic domain, and then has that same party send masked values to the remaining parties, enabling them to adjust their shares accordingly. As a result, the identical underlying bit is encoded in $\mathbb{F}_{2^{32}}$. Our code uses MultiChor's `locally` combinator to describe party-local calculations, the `~>` operator to manage inter-party communication, and our `locs2Faceted` helper to combine the final triple of shares into a `Faceted` value.

Algorithm 5: Protocol $[\![v]\!] \leftarrow$ ShareConv($[\![u]\!]$) for converting a share $[\![u]\!] \in \mathbb{Z}_2$ to $[\![v]\!] \in \mathbb{Z}_{2^n}$.

Data: Shared value $[\![u]\!]$ in bit shares.
Result: Shared value $[\![v]\!]$ such that $u = v$ and $[\![v]\!]$ is shared in $\mathbb{Z}_{2^n}$.

1 $\mathcal{P}_1$ generates random $b \leftarrow \mathbb{Z}_2$ and sets $m \leftarrow b \oplus u_1$.
2 $\mathcal{P}_1$ locally converts m to $\mathbb{Z}_{2^n}$, generates random $m_{12} \leftarrow \mathbb{Z}_{2^n}$ and computes $m_{13} = m - m_{12}$.
3 $\mathcal{P}_1$ generates random $b_{12} \leftarrow \mathbb{Z}_2$ and computes $b_{13} = b - b_{12} = b \oplus b_{12}$.
4 All values $*_{ij}$ are sent from $\mathcal{P}_i$ to $\mathcal{P}_j$.
5 $\mathcal{P}_2$ sets $s_{23} \leftarrow b_{12} \oplus u_2$.
6 $\mathcal{P}_3$ sets $s_{32} \leftarrow b_{13} \oplus u_3$.
7 All values $*_{ij}$ are sent from $\mathcal{P}_i$ to $\mathcal{P}_j$.
8 $\mathcal{P}_1$ sets $v_1 \leftarrow 0$
9 $\mathcal{P}_2$ and $\mathcal{P}_3$ set $s \leftarrow s_{23} \oplus s_{32}$
10 **if** $s = 1$ **then**
11 $\quad \mathcal{P}_2$ sets $v_2 \leftarrow (1 - m_{12})$.
12 $\quad \mathcal{P}_3$ sets $v_3 \leftarrow (-m_{13})$.
13 **else**
14 $\quad \mathcal{P}_2$ sets $v_2 \leftarrow m_{12}$.
15 $\quad \mathcal{P}_3$ sets $v_3 \leftarrow m_{13}$.
16 **end**
17 Return $[\![v]\!]$.

Fig 3.23 - ShareConv pseudocode [39]

In our implementation of the BitExtraction protocol, we used Algorithm 6 from Bogdanov et al.[39], whose goal is to represent the shared input u as $v + r$, where r is a random number whose bitwise decomposition is already known by the parties. By adding v and r and utilizing the CarryBits function, the parties identify the carry bits that occur during addition.Since Bogdanov et al.'s paper does not specify the CarryBits function, we build it using the effective 1-bit adder from[48], shown in Figure 3.24 which computes the carry-out bit as

$$c_{i+1} = c_i \oplus ((x_i \oplus c_i) \wedge (y_i \oplus c_i))$$

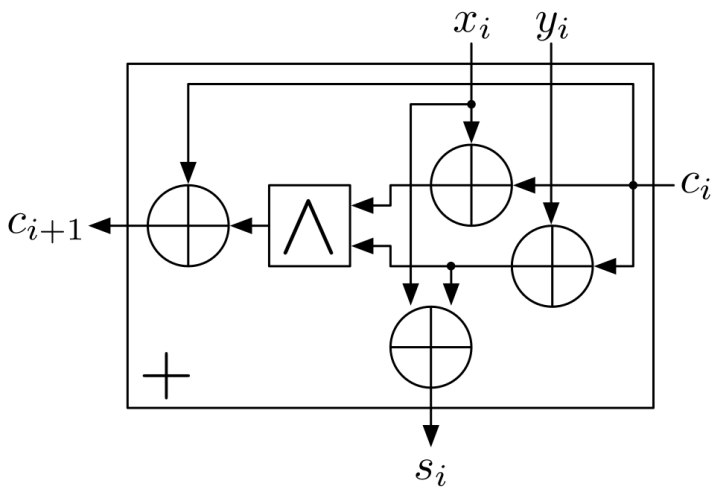

Fig 3.24 - 1-bit Adder (+) [48]

To make working with bit arrays and their operations practical in Haskell, we provide helper functions to convert between `Faceted ps '[] [Bool]` and `[Faceted ps '[] Bool]` and vice versa. These conversions are the basic machinery of the `SList` class introduced in Section 3.2.1, and allow secure functions to be applied to lists using standard higher-order combinators such as `map` and `fold`. Lastly, the parties can reconstruct the binary representation of u without disclosing its value by combining the securely computed carry bits with the known (but secret-shared) bits of r.

While Bogdanov et al.[39] specify ShareConv $(\mathbb{F}_2 \to \mathbb{F}_{2^{32}})$ and BitExtraction $(\mathbb{F}_{2^{32}} \to \mathbb{F}_2^{32})$, they do not include the inverse of BitExtraction $(\mathbb{F}_2^{32} \to \mathbb{F}_{2^{32}})$. To implement this operation, we followed the protocol XorToAdd from Laud et al [49] presented in the Figure 3.25 where the ideia is straightforward:

1. Conversion of every bit: Use ShareConv to convert each secret bit to the arithmetic sharing domain $\mathbb{F}_{2^{32}}$;
2. Apply the respective weights: Multiply each converted value by its positional weight 2^i where i is the bit index (we use little-endian order, so for our case b_0 has weight 1, b_1 has weight 2, etc.);
3. Sum the results: To determine the final arithmetic share, securely sum up all of the weighted values.

Algorithm 1: XorToAdd protocol

Data: $n \in \mathbb{N}$, and shared bits $[\![b_0]\!], \ldots, [\![b_{n-1}]\!]$
Result: $[\![m]\!]$, shared over $\mathbb{Z}_{2^n}$, such that $m = \sum_{i=0}^{n-1} 2^i b_i$
1 Parties use the protocol ShareConv to convert each $[\![b_i]\!]$ from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^n}$, getting n additively shared values c_i.
2 **Return** the additively shared value $[\![m]\!] = \sum_{i=0}^{n-1} 2^i [\![c_i]\!]$.

Fig 3.25 - XorToAdd pseudocode [49]

In our implementation, we replicate this structure: Utilizing `mapM`, we first apply ShareConv to every element of the `[Faceted ps '[] Bool]` input, transforming every bit from boolean to arithmetic sharing. The positional weights are then calculated in little-endian order as `[2^i :: a | i <- [0..n-1]]` and we use `zipWithM` and `mulConst` to associate each converted bit with its corresponding weight. Finally, to compute the sum we fold the earlier zipped bit vector with the secure addition function starting from a securely classified zero resulting in a `Faceted` value representing the reconstructed number in $\mathbb{F}_{2^{32}}$

Equality and Ordering Functions To evaluate secret equalities and comparisons, Sharemind offers two crucial building blocks: Equal and GTE (Greater-than-or-equal). Figure 3.26 shows how we implemented Equal in accordance with Algorithm 7 from Bogdanov et al. [39]. The protocol masks the difference $u - v$ with random values, allowing parties $\mathcal{P}_2$ and $\mathcal{P}_3$ to get additive shares e_2 and e_3 of the difference without leaking anything about u or v. The next step is to determine whether $e_2 + e_3 = 0$, which is accomplished bitwise: $\mathcal{P}_1$ assigns its share to an all-ones bitstring, whereas $\mathcal{P}_2$ holds bin(e_2) and $\mathcal{P}_3$ holds bin$(-e_3)$. This works because the bitwise XOR of these three values yields an all-ones bitstring, if and only if $u - v = 0$, since in that case e_2 and $-e_3$ are the same value, their XOR produces all zeros, and XOR'ing with $\mathcal{P}_1$ all-ones string restores all ones. Last but not least, BitConj safely calculates the conjunction of every bit in this string (executed as a fold using the secure multiplication gate over $\mathbb{F}_2$), yielding a secret-shared Boolean that is `True` if the two inputs are equal.

Algorithm 7: Protocol $[\![w]\!] \leftarrow$ Equal($[\![u]\!], [\![v]\!]$) for evaluating the equality predicate.

Data: Shared values $[\![u]\!]$ and $[\![v]\!]$.
Result: Shared value $[\![w]\!]$ such that $w = 1$ if $u = v$, and 0 otherwise.

1 $\mathcal{P}_1$ generates random $r_2 \leftarrow \mathbb{Z}_{2^n}$ and computes $r_3 \leftarrow (u_1 - v_1) - r_2$.
2 $\mathcal{P}_1$ sends r_i to $\mathcal{P}_i$ $(i = 2, 3)$.
3 $\mathcal{P}_i$ computes $e_i = (u_i - v_i) + r_i$ $(i = 2, 3)$.
4 $\mathcal{P}_1$ sets $\overline{p}_1 \leftarrow 2^n - 1 = 111 \ldots 1$.
5 $\mathcal{P}_2$ sets $\overline{p}_2 \leftarrow e_2$.
6 $\mathcal{P}_3$ sets $\overline{p}_3 \leftarrow (0 - e_3)$.
7 Return $[\![w]\!] \leftarrow$ BitConj($\overline{[\![p]\!]}$).

Fig 3.26 - Equals pseudocode [39]

We use Sharemind's Algorithm 5 from the Sharemind paper [35], to the greater-than-or-equal predicate, which reduces the comparison to a sign-bit test on the difference. In practical terms, each party first calculates an additive sharing of $d = u - v$, and then they all perform BitExtraction to get the bit decomposition of d. The most significant bit from each party's share of d is then extracted and inverted.

Using these two fundamental building blocks, the remaining ordering operations can be written as straightforward combinations of Equal and GTE with simple Boolean gates like `not` and `or`, both of which are implemented precisely as they are in our MPC interface's default instance. In particular, `less-than (lt)` is the negation of GTE since $\neg(\geq)$ is equal to $<$. `less-than-or-equal (lte)` is obtained by computing `lt` and Equal, and then applying the `or` gate to their outputs and lastly, `greater-than (gt)` is just the negation of `lte`.

Bit-level operations This section discusses Sharemind's standard bit-level operations on secret-shared values, aside from conversion protocols. We emphasize on more complex and useful primitives, such as the MSNZB protocol, which identifies the position of a shared value's most significant nonzero bit, along with bit shifts (left and right) and many more auxiliary sub-protocols that are utilized in their construction.

We start with the PrefixOR protocol, which is an essential primitive of the MSNZB protocol. PrefixOR operates intuitively by taking a bitwise secret-shared vector as input and producing a different bitwise vector with the most significant positive bit "propagated" downward. For instance, in our little-endian setup, the prefix-or of the 8-bit number 00101100_2 is 00111111_2. Formally speaking, according to the definition in Laud and Randmets [45], if

$u = (u_1, \ldots, u_n)$ indicates the bit decomposition of a value, then $v = (v_1, \ldots, v_n)$ is defined by $v_i = \bigvee_{j=i}^{n} u_j$, with each position i storing the OR of all more significant bits from u.

We follow the recursive definition of the PrefixOR method as described in Algorithm 3 (Figure 3.27) from [39]. Our implementation takes a bitwise-shared vector in the format `[Faceted ps '[] Bool]` and using the auxiliary functions `firsthalf` and `secondhalf`, the input is separated into two halves, and PrefixOR is invoked recursively on each. When both recursive calls return, the results are combined into a temporary vector `p'`. Following lines 7-8 of the pseudocode, we then construct a new array `p` by applying `mapM` with the `or` gate to combine each element of the left half, `p' !! i`, with the most significant element of the right half, `p' !!` $\lfloor l/2 \rfloor$, for all indices `i <- [0..`$\lfloor l/2 \rfloor$`-1]`. Finally, we remove the first $\lfloor l/2 \rfloor$ elements from `p'` (`d = drop l2 p'`) and return the concatenation `p ++ d`, which represents the updated left half followed by the unmodified right half.

Algorithm 3: $\overline{[\![p']\!]} \leftarrow \text{PrefixOR}(\overline{[\![p]\!]})$.

Data: Bitwise shared vector $\overline{[\![p]\!]}$.
Result: The vector $\overline{[\![p']\!]}$ which has the form $00\ldots011\ldots1$, where the initial part $00\ldots01$ coincides with the vector originally represented by $\overline{[\![p]\!]}$.

1 $l \leftarrow |\overline{[\![p]\!]}|$.
2 **if** $l = 1$ **then**
3 Return $\overline{[\![p']\!]} \leftarrow \overline{[\![p]\!]}$.
4 **else**
5 $\overline{[\![p']\!]}^{(l-1\ldots\lfloor l/2 \rfloor)} \leftarrow \text{PrefixOR}(\overline{[\![p]\!]}^{(l-1\ldots\lfloor l/2 \rfloor)})$.
6 $\overline{[\![p']\!]}^{(\lfloor l/2 \rfloor-1\ldots0)} \leftarrow \text{PrefixOR}(\overline{[\![p]\!]}^{(\lfloor l/2 \rfloor-1\ldots0)})$.
7 **for** $i \leftarrow 0$ **to** $\lfloor l/2 \rfloor - 1$ **do** .
8 $\overline{[\![p']\!]}^{(i)} \leftarrow \overline{[\![p']\!]}^{(i)} \vee \overline{[\![p']\!]}^{(\lfloor l/2 \rfloor)}$.
9 Return $\overline{[\![p']\!]}$.
10 **end**

Fig 3.27 - PrefixOR pseudocode [39]

Based on this, we build the MSNZB protocol as defined in Algorithm 4 (Figure 3.28) from [39]. We first compute the input vector's PrefixOR value and assign its output to a variable `u'`. Next, we follow lines 2-3 of the pseudocode, where we construct an intermediate array by applying `mapM` with the `xor` operation (implemented through our `add` gate) to each pair of consecutive elements, `u' !! i` and `u' !! (i+1)`, for all indices `i <- [0..nBits-2]`. Lastly, as indicated in line 4 of Algorithm 4, we append the most significant element of `u'` (`u' !! (nBits-1)`) to this result, yielding the output vector which denotes the most significant nonzero bit position.

Algorithm 4: Protocol for the most significant nonzero bit position $\overline{[\![s]\!]} \leftarrow \mathsf{MSNZB}(\overline{[\![u]\!]})$.

Data: Bitwise shared value $\overline{[\![u]\!]}$.
Result: Shared vector $\overline{[\![s]\!]}$ such that $\overline{[\![s]\!]}^{(j)}$ represents 1, where j is the most significant position, where $\overline{[\![u]\!]}^{(j)}$ represents 1, and 0 otherwise. If all the bits of $\overline{[\![u]\!]}$ represent 0, all the shared bits of $\overline{[\![s]\!]}$ represent 0 as well.

1 $\overline{[\![u']\!]} \leftarrow \mathsf{PrefixOR}(\overline{[\![u]\!]})$.
2 **for** $i \leftarrow 0$ **to** $n-2$ **do**
3 $\quad \overline{[\![s]\!]}^{(i)} \leftarrow \overline{[\![u']\!]}^{(i)} \oplus \overline{[\![u']\!]}^{(i+1)}$.
4 $\overline{[\![s]\!]}^{(n-1)} \leftarrow \overline{[\![u']\!]}^{(n-1)}$.
5 Return $\overline{[\![s]\!]}$.

Fig 3.28 - MSNZB pseudocode [39]

We now move on to bit-shift protocols with a public shift, which are fundamental primitives in the development of Sharemind's secure-division protocols. We start with the left-shift operation where a left shift by p positions can be realized by multiplying the shared value by the public constant 2^p, which in practice amounts to locally multiplying each share by the same constant. Our implementation, however, relies on the Haskell's `shiftL :: forall a. Bits a => a -> Int -> a`, provided by `GHC.Internal.Bits`, since all fixed-width word types provide `Bits` instances via the `Data.Word` module.

On the other hand, the right-shift protocol is not so straightforward since a right shift by p corresponds to division by 2^p in $\mathbb{Z}_{2^n}$, and correctness depends on the remainder bits that are discarded during the shift. Following Bogdanov et al. [39], Sharemind handles this by first resharing the value to two parties (using the previously mentioned reshareToTwo protocol), so that $[[u']] = (u'_1, u'_2, u'_3)$ with $u'_1 = 0$. On this two-party view, we compute the Overflow bit via Algorithm 11:

$$\lambda \leftarrow \mathbf{Overflow}([[u']]) \quad \text{with} \quad \lambda = \begin{cases} 1 & \text{if } u'_2 + u'_3 \geq 2^n \ \ (\text{equiv. } u'_2 \geq 2^n - u'_3), \\ 0 & \text{otherwise,} \end{cases}$$

i.e., λ indicates whether addition of the two shares wrapped around modulo 2^n. This is accomplished by using Algorithm 11, as seen in Figure 3.29 from [39]. The primary idea is to compare u'_2 with $-u'_3$ (mod2n) by invoking MSNZB, which returns a bitwise vector that is all zeroes if $u'_2 = -u'_3 \pmod{2^n}$, or has a single 1 marking the most significant position where the two values differ. $\mathcal{P}_3$ then shares its value $-u'_3$ bitwise with the other parties, allowing for the computation of a dot product between the vector and the MSNZB output. According to [39], the dot product equals 1 if and only if $u'_2 < u'_3 \pmod{2^n}$. As a result,

$\lambda 0 = 1$ implies that $u'_2 \geq -u'_3 \pmod{2^n}$, indicating the desired overflow scenario. To handle the special case $u'_3 = 0$, $\mathcal{P}_3$ locally flips its share of $\lambda 0$ and the result is transformed back into an arithmetic share with ShareConv. In our implementation, the dot product is implemented as `mu <- zipWithM mult s' u3list` followed by `foldM add2Shares acc mu`.

Algorithm 11: Protocol $[\![\lambda]\!] \leftarrow$ Overflow$([\![u']\!])$ for obtaining the overflow bit $[\![\lambda]\!]$ for $[\![u']\!]$ if share $u'_1 = 0$.

Data: Shared value $[\![u']\!]$ where $u'_1 = 0$.
Result: Share $[\![\lambda]\!]$ so that $u' = u'_2 + u'_3 - \lambda 2^n$.

1 $\mathcal{P}_1$ sets $p_1 = 0$.
2 $\mathcal{P}_2$ sets $p_2 = u'_2$.
3 $\mathcal{P}_3$ sets $p_3 = -u'_3$.
4 $\overline{[\![s]\!]} \leftarrow$ MSNZB$(\overline{[\![p]\!]})$.
5 Share the value $-u'_3$ bitwise as a vector $\overline{[\![-u'_3]\!]}$.
6 $[\![\lambda^0]\!] \leftarrow 1 \oplus \bigoplus_{i=0}^{n-1} \overline{[\![s]\!]}^{(i)} \wedge \overline{[\![-u'_3]\!]}^{(i)}$.
7 $\mathcal{P}_3$ checks whether $u'_3 = 0$. If so, $\lambda_3^0 = 1 \oplus \lambda_3^0$.
8 Return $[\![\lambda]\!] \leftarrow$ ShareConv$([\![\lambda^0]\!])$.

Fig 3.29 - Overflow pseudocode [39]

With the overflow protocol defined, we can now define the right-shift operation (Algorithm 12). The idea is to reshare the input into two parties, shift each part locally, and then adjust the result using the overflow bits, as shown in Figure 3.30. Two corrections are needed: (i) the carry bit discarded from the least significant positions, handled by computing `λ1 = Overflow(u')` (line 3), whose effect is corrected in line 6 by subtracting $2^{n-p}[[\lambda_1]]$; and (ii) the top carry bit that typically vanishes under modulo-2^n arithmetic but returns when the values are shifted down, detected via `λ2 = Overflow(s)` (lines 2-4) and added back in line 6 which ensures that the output equals $\lfloor u/2^p \rfloor \pmod{2^n}$.

Algorithm 12: Protocol $[\![w]\!] \leftarrow$ ShiftR$([\![u]\!], p)$ for evaluating right shift.

Data: Shared value $[\![u]\!]$ and a public shift p.
Result: Shares $[\![w]\!]$ such that $w = u \gg p$.

1 $[\![u']\!] \leftarrow$ ReshareToTwo$([\![u]\!])$.
2 $[\![s]\!] \leftarrow [\![u' \ll n - p]\!]$ (locally).
3 $[\![\lambda_1]\!] \leftarrow$ Overflow(u').
4 $[\![\lambda_2]\!] \leftarrow$ Overflow(s).
5 $\mathcal{P}_i$ computes $v_i \leftarrow u'_i \gg p$.
6 Return $[\![w]\!] = [\![v]\!] - 2^{n-p}[\![\lambda_1]\!] + [\![\lambda_2]\!]$.

Fig 3.30 - ShiftR pseudocode [39]

Division protocols Finally, we turn to the two division protocols provided by Sharemind: one for dividing a secret-shared dividend by a public divisor, Algorithm 8: PubDiv, and another for the case where both dividend and divisor are secret-shared, Algorithm 9: Div. Both protocols require intermediate computations on values larger than Sharemind's default 32-bit domain and since these values must also be reduced modulo 2^k for a parameterized bit-length k, we could not rely on the standard `Word` types. Instead, we adopted the library `Data.BitVector.Sized`, which provides fixed-size bit-vectors with type-level sizes. This choice allows us to work uniformly with values of arbitrary bit-width while ensuring that all arithmetic is reduced modulo 2^k by construction. To interface with existing code and host types, we defined a small adapter class, as shown in Figure 3.31

```
1 -- words of size @n@, with numbers ranging [0..2^n-1]
2 class WordSized t n where
3     toBV :: t -> BV n
4     fromBV :: BV n -> t
```

Fig 3.31 - WordSized Class

which we instantiate for common types (e.g., `Bool`, `Word32`, `Word64`) and selected larger widths as needed. This abstraction makes conversions explicit and type-safe, while keeping our protocol code generic over the bit-width.

Given this, we now focus on the division protocols. In the case of a public divisor, the division protocol, denoted PubDiv in [39] and illustrated in Figure 3.32, reduces the operation to a multiplication by a precomputed constant. The idea, originating from prior work [50], is to approximate the reciprocal of the public divisor d by an integer of the form

$$d' \approx c \cdot 2^{-k},$$

where c and k are chosen so that the result of the multiplication

$$\left\lfloor \frac{u}{d} \right\rfloor = \left\lfloor \frac{c \cdot u}{2^k} \right\rfloor$$

holds for all inputs u in $\mathbb{Z}_{2^n}$. In practice, taking $k = n + 1$ is sufficient, since we are working with unsigned integers in $\mathbb{Z}_{2^{32}}$ and the problem reduces to securely computing the product $c \cdot u$ followed by a right-shift by k bits.

Because multiplying two n-bit values may produce up to a $2n$-bit result, the protocol first promotes the operands to $2n+1$ bits , performs the multiplication, and then truncates the result back to n bits. To handle the potential carry into the higher bits during truncation, two candidate results are prepared, where one assuming no carry and the other assuming a carry. Finally, both are shifted right by k, and the secure Overflow protocol is invoked to decide which candidate is correct.

Algorithm 8: Protocol $[\![w]\!] \leftarrow \mathsf{PubDiv}([\![u]\!], d)$ for division by a public value d.

Data: Shared value $[\![u]\!]$ and a public divisor d.
Result: Shares $[\![w]\!]$ of the value $w = \lfloor \frac{u}{d} \rfloor$.

1 $[\![u']\!] \leftarrow \mathsf{ReshareToTwo}(u)$.
2 $\mathcal{P}_2, \mathcal{P}_3$ find $c \in \mathbb{Z}_{2^{n+1}}$ such that $c2^{-(n+1)} \approx \frac{1}{d}$ as in [13].
3 $\mathcal{P}_1$ sets $v_1^1, v_1^2 \leftarrow 0 \in \mathbb{Z}_{2^{2n+1}}$.
4 $\mathcal{P}_2$ sets $v_2^1 \leftarrow cu_2', v_2^2 \leftarrow c(u_2' - 2^n) \in \mathbb{Z}_{2^{2n+1}}$.
5 $\mathcal{P}_3$ sets $v_3^1, v_3^2 \leftarrow cu_3' \in \mathbb{Z}_{2^{2n+1}}$.
6 $[\![\lambda]\!] \leftarrow \mathsf{Overflow}([\![u']\!])$.
7 $[\![\lambda^1]\!] \leftarrow \mathsf{Overflow}([\![v^1 \ll n]\!])$.
8 $[\![\lambda^2]\!] \leftarrow \mathsf{Overflow}([\![v^2 \ll n]\!])$.
9 Every $\mathcal{P}_i$ sets $w_i^1 \leftarrow v_i^1 \gg (n+1)$ and $w_i^2 \leftarrow v_i^2 \gg (n+1)$.
10 Return $[\![w]\!] \leftarrow (1 - [\![\lambda]\!])([\![w^1]\!] + [\![\lambda^1]\!]) + [\![\lambda]\!]([\![w^2]\!] + [\![\lambda^2]\!])$.

Fig 3.32 - PubDiv pseudocode [39]

Finally, we consider the protocol for dividing two secret shared values. Sharemind implements division using the Goldschmidt iteration method, formalized as Div in [39] and illustrated in Figure 3.33. Goldschmidt's algorithm is an adaptation of Newton iteration designed for efficient division in digital computers: starting from an initial approximation of the reciprocal of the divisor, it iteratively refines the quotient by maintaining scaled values N_i and D_i such that $N_i D_i = uv$ with $D_i \to 1$ as i grows, ensuring $N_i \to u/v$. However, adapting this method to Sharemind introduces challenges, since it is most often used for floating-point division, Sharemind emulates fix-point arithmetic by converting everything to purely integer arithmetic with corresponding integer operations followed by the appropriate bit shifts. Since modulus expansion and an efficient shift right operators and truncations are essential, Sharemind circunvents this by expanding its ring to m bits so that all multiplications can be carried out in a sufficiently large domain without repeated expansions, and by applying imprecise truncation by shifting shares individually without correcting for carry bits. Each iteration then updates the pair (N_i, D_i) in parallel by computing a scaling factor

$$F_i = 2 - D_{i-1}, \quad N_i = F_i N_{i-1}$$

and

$$D_i = F_i D_{i-1},$$

and thanks to the quadratic convergence of Goldschmidt iteration, only $\log_2 n$ steps are required to achieve n bits of precision. To offset this loss of accuracy of imprecise truncation during the iteration, Sharemind slightly increases the working precision and modifies the first iteration with a stronger update rule, ensuring that convergence remains monotonic from below. Finally, before truncation back to the original domain, the protocol adds a small correction term Δ to the result $N_{\log_2 n+1}$, guaranteeing strict downward rounding of the quotient, ensuring that the final output always under-approximates the true division result.

Algorithm 9: Protocol $[\![w]\!] \leftarrow \mathsf{Div}([\![u]\!], [\![v]\!])$ for division.

Data: Shared values $[\![u]\!]$ and $[\![v]\!]$.
Result: Shares $[\![w]\!]$ such that $w = \lfloor \frac{u}{v} \rfloor$.

1 $\overline{[\![v']\!]} \leftarrow \mathsf{BitExtr}([\![v]\!])$.
2 $\overline{[\![s]\!]} \leftarrow \mathsf{MSNZB}(\overline{[\![v']\!]})$.
3 Compute $[\![c^i]\!] \leftarrow \mathsf{ShareConv}^{\mathsf{m}}(\overline{[\![s]\!]}^{(i)})$ $(i = 0, \ldots, n-1)$.
4 Set $[\![\widehat{c_0}]\!] \leftarrow \sum_{i=0}^{n-1} 2^{n'-i-1}[\![c^i]\!]$.
5 $[\![u']\!] \leftarrow \mathsf{ReshareToTwo}([\![u]\!])$, $[\![\lambda^1]\!] \leftarrow \mathsf{Overflow}^{\mathsf{m}}([\![u']\!])$.
6 $[\![v']\!] \leftarrow \mathsf{ReshareToTwo}([\![v]\!])$, $[\![\lambda^2]\!] \leftarrow \mathsf{Overflow}^{\mathsf{m}}([\![v']\!])$.
7 $[\![u'']\!] \leftarrow [\![u']\!] - 2^n[\![\lambda^1]\!] \in \mathbb{Z}_{2^m}$.
8 $[\![v'']\!] \leftarrow [\![v']\!] - 2^n[\![\lambda^2]\!] \in \mathbb{Z}_{2^m}$.
9 Compute $[\![\widehat{N_0}]\!] \leftarrow [\![u'']\!] \cdot [\![\widehat{c_0}]\!]$ and $[\![\widehat{D_0}]\!] \leftarrow [\![v'']\!] \cdot [\![\widehat{c_0}]\!]$.
10 Set $[\![\widehat{F_1}]\!] \leftarrow \lfloor 2^{n'} \cdot 2\sqrt{2} \rfloor - 2[\![\widehat{D_0}]\!]$.
11 Compute $[\![\widehat{\widehat{N_1}}]\!] \leftarrow [\![\widehat{N_0}]\!] \cdot [\![\widehat{F_1}]\!]$ and $[\![\widehat{\widehat{D_1}}]\!] \leftarrow [\![\widehat{D_0}]\!] \cdot [\![\widehat{F_1}]\!]$.
12 **for** $k \leftarrow 1$ **to** $\log_2 n$ **do**
13 Each party $\mathcal{P}_i$ computes $(\widehat{N_k})_i \leftarrow ((\widehat{\widehat{N_k}})_i \gg n') + 1$ and $(\widehat{D_k})_i \leftarrow (\widehat{\widehat{D_k}})_i \gg n'$ $(i = 1, 2, 3)$.
14 Set $[\![\widehat{F_{k+1}}]\!] \leftarrow 2^{n'} \cdot 2 - [\![\widehat{D_k}]\!]$.
15 Compute $[\![\widehat{\widehat{N_{k+1}}}]\!] \leftarrow [\![\widehat{N_k}]\!] \cdot [\![\widehat{F_{k+1}}]\!]$ and $[\![\widehat{\widehat{D_{k+1}}}]\!] \leftarrow [\![\widehat{D_k}]\!] \cdot [\![\widehat{F_{k+1}}]\!]$.
16 Compute $[\![R]\!] \leftarrow [\![\widehat{\widehat{N_{\log_2 n+1}}}]\!] + \Delta$.
17 Return $[\![w]\!] \leftarrow \mathsf{ShiftR}^{n+n'}([\![R]\!], n') \bmod 2^n$.

Fig 3.33 - Div pseudocode [39]

While we successfully implemented the remaining Sharemind arithmetic primitives, we were unable to complete Algorithm 9 Div within the given time frame. The primary challenge is managing the Goldschmidt iteration's cost and precision in an MPC setting. The protocol relies on expensive bit-level operations to determine the initial scaling constant, requires careful handling of modulus expansion, and features imprecise truncation steps whose

error must be compensated consistently across iterations. In our attempts, even small inaccuracies in truncation and rescaling introduced nondeterminism, with the same inputs producing different (incorrect) results. Since secure division is one of the most complex arithmetic operations in MPC, we leave a correct and efficient implementation of this protocol for future work, and instead focus our evaluation on the arithmetic primitives we were able to realize in our eDSL.

To summarize, our implementation displays how Sharemind primitives connect to the abstract functions of our MPC interface. For instance, the secure multiplication operation in the interface is directly mapped to Sharemind's native multiplication protocol, while other arithmetic, conversion, comparison, and ordering operations are similarly linked to their backend counterparts. The only function not overridden is the secure multiplexer (`mux`), as Sharemind does not provide a dedicated protocol for this gate. In this case, we rely on the default instance from the interface.

3.3.2 Instantiation of the Motion Backend

We now move on to the second backend instance of our eDSL: the Motion backend. As previously stated, Motion is a framework for mixed-protocol N-party MPC; however, in this thesis, we limited ourselves to its linear secret sharing schemes and focused solely on the arithmetic GMW protocol. Following the structure of the Sharemind instantiation, we begin by outlining the processes for sharing and classifying values, as well as unsharing and declassifying them in this N-party environment. We then highlight Motion's multiplication protocol, before moving on to the remaining operations, such as conversions, equality, and multiplexing, and demonstrating how each is connected to our interface's abstract functions.

The additive secret sharing mechanism underlying Motion's arithmetic GMW backend is very similar to Sharemind's approach. While Sharemind is set at additive shares over 32-bit unsigned integers in a three-party setting, arithmetic GMW is more flexible, as it may be instantiated over a ring of l-bit integers and extended to an arbitrary n number of parties. In this paradigm, a secret value x is represented by n shares $(x_1, x_2, \ldots, x_n)$ so that

$$x_1 +' x_2 +' \cdots +' x_n \equiv x \pmod{2^l}.$$

Given this, the creation of shares and the distribution mechanism in Motion adhere to the same concepts as Sharemind, but in a n-party environment. Our adaptation of Sharemind's

`secretShare` works as follows: The input party generates $n-1$ random l-bit values and then calculates the final share as

$$x_n \equiv x -' (x_1 +' x_2 +' \cdots +' x_{n-1}) \pmod{2^l}.$$

As before, the resulting vector of shares is distributed to the computing parties using MultiChor's `scatter` function, ensuring that each participant receives just their own local share.

Classification of shares in the Motion backend required a slightly different strategy. In Sharemind's fixed three-party architecture, categorization is simple: any public value x can potentially be encoded deterministically as $(x, -x, x)$, as these shares always sum up to x modulo 2^l. For Motion's general N-party configuration, however, we created a technique that adjusts to the party set's parity. Specifically, we first establish whether the number of computing parties is odd or even.

- Odd case (e.g., $n = 5$): we assign the shares as $(x, x, x, -x, -x)$. Here, the positive and negative components cancel out, leaving x as the reconstructed value.
- Even case (e.g., $n = 4$): we introduce one zero share and then apply the same logic as above, $(0, x, -x, x)$.

This construction guarantees that classification of values remains deterministic and that the aggregate of all shares always matches the underlying value, regardless of the number of computing participants.

For declassifying and unsharing in Motion, we follow the same strategy as in the Sharemind backend. MultiChor's `gather` function first collects shares from a `Faceted` value and delivers them to the designated receivers (the computing parties for declassification, or the output parties for unsharing). The `congruently` function then spreads the resultant `Quire` uniformly between all parties, adding up the shares to reconstruct the secret. The result is a `Located` value: in the unsharing scenario, this value sits at the output parties; in the declassification situation, it is transformed to a clear value using MultiChor's `naked` function.

Unlike Sharemind, the Motion backend does not have a specific `reshare` primitive since its multiplication protocol uses another method. Consequently, we didn't include a corresponding function in our MPC framework. As a result, our Motion instance only implements

the default generic definition supplied by the MPC interface to implement the more general `reshareTo` operation. However, since `reshareTo` also supports the case where the source and target coalitions are the same, it effectively covers the role of `reshare` in this backend. Following the same pattern as before, we discuss Motion's support for basic arithmetic operations (addition, multiplication, etc.) before moving on to its conversion functions and the remaining operations required by our MPC interface.

Core Arithmetic Primitives Similarly to Sharemind, Motion also has operations that can be carried out locally. Addition, subtraction, negation, and multiplication by a constant are implemented in the same way, as they fit naturally within the structure of linear secret sharing schemes.

Multiplication between two secret-shared values, however, is very different, as it is based on Beaver Multiplication Triples (MTs) [28]. These triples are generated using a generalized additively correlated oblivious transfer protocol (C^+-OT) [37], which extends standard oblivious transfer to produce additive correlations suitable for arithmetic sharing. In the N-party context, the diagonal terms can be computed locally, while the cross-terms of the product are computed interactively by each pair of parties. This produces additive shares of random values a,b, and their product $c = a *' b$. Typically, these triples are generated in bulk during the protocol's offline phase and consumed once during the online phase as needed.

With this, Given a secret-shared MT $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ such that $a *' b \equiv c \pmod{2^l}$ and two secret shared inputs, $\langle x\rangle$ and $\langle y\rangle$, the multiplication protocol proceeds as follows: First, the parties reconstruct the masked differences

$$d \leftarrow \text{Declassify}(\langle x\rangle -' \langle a\rangle), \quad e \leftarrow \text{Declassify}(\langle y\rangle -' \langle b\rangle)$$

which become public values. They then compute the result as

$$\langle z\rangle \leftarrow \langle c\rangle +' e *' \langle x\rangle +' d *' \langle y\rangle -' d *' e,$$

which yields the desired product $\langle z\rangle \leftrightarrow \langle x *' y\rangle$.

To simplify our implementation, we abstracted away both the offline/online phase distinction and the actual C^+-OT creation of triples, instead modeling a trusted third party that produces triples on-demand. This decision allows for a natural expression of our encoding as a choreography while maintaining faithfulness to the protocol's structure. Although the triple

generator is trusted in our current implementation for simplicity, it might be replaced in further work with a proper secure offline protocol for generating triples, thereby recovering the conventional offline/online separation used in MPC.

The declassification step's intermediate values, d and e, don't reveal anything about the underlying secrets. This is because $d = x -' a$ and $e = y -' b$ are masked by the random shares a and b taken from the Beaver triple. The values d and e are statistically independent of x and y since a and b are uniformly random and unknown to the computing parties before declassification. Therefore, declassification only exposes the secret differences, which are necessary for the secure multiplication procedure and safe to disclose to all the computing parties.

Given this, whenever a multiplication is called, this generator creates random numbers $a, b, c = a *' b$ and secretly distributes them among the computing parties, as illustrated in Figure 3.34.

```
getBeaverTriple ::
  forall a m ps psr.
  (ps ∼ ("trustedgen" : psr), Random a) =>
  Choreo ps m (Faceted ps '[] a, Faceted ps '[] a, Faceted ps '[] a)
getBeaverTriple = do

  let trustedall = listedFirst :: Member "trustedgen" ps

  aT <- trustedall `locally` \_ -> liftIO (randomIO :: IO a)
  bT <- trustedall `locally` \_ -> liftIO (randomIO :: IO a)

  cT <- trustedall `locally` \un -> do
    let aT' = un singleton aT
        bT' = un singleton bT
        cT = aT' *' bT'
    return (cT)

  aShares <- secretShare trustedall refl aT
  bShares <- secretShare trustedall refl bT
  cShares <- secretShare trustedall refl cT

  return (aShares, bShares, cShares)
```

Fig 3.34 - Beaver triplets generation

However, because our interface is defined as shown in Figure 3.35 the multiplication must type-check the set `ps` of computing parties. As a result, the trusted generator must formally appear in `ps`. However, this generator is not a proper computing party since it should never own actual shares of the inputs. To accomplish this, we modified Motion's `classify` and `share` functions to always give the trusted party zero shares. In this way, it belongs to `ps` at the type level but does not contribute to the computation. This method

is conceptually identical to modeling the absence of a share with a type like `Maybe a`; for convenience, we we chose the simpler zero-share representation.

```
class (MPC protocol m ps a, Num a) => SNum protocol m ps a where
  ...
  mul :: Secret protocol ps a -> Secret protocol ps a -> Choreo ps m (Secret
    protocol ps a)
  ...
```

Fig 3.35 - Interface mul type signature

Share Conversion Functions To convert Arithmetic shares to Boolean shares and vice versa, we followed Motion's `A2B` and `B2A` protocols, which makes use of Extended Shared Bits (ESBs). These ESBs, similar to multiplication triples, are extended to tuples of the form $(\langle r \rangle^B, \langle r \rangle^A)$ where (A) and (B) represent the arithmetic and Boolean domains, respectively. The value r is represented as:

$$r = \sum_{k=0}^{l-1} 2^k \cdot r_k$$

and is used to optimize the conversion protocols. These ESBs are originally known as extended doubly-authenticated bits (edaBits) in active secure MPC settings, but Motion and our implementation simply offer security against semi-honest adversaries and do not require authentication. Just like multiplication triples, ESBs can be precomputed offline, but in our adaptation, ESBs are computed on demand by our trusted third party.

In our implementation, the trusted third party generates a random arithmetic value r^A, converts it to its binary representation r^B, yielding `Located '[T] a` and `[Located '[T] Bool]` respectively, where T denotes the trusted third party generator. We then use Motion's secret-sharing functions, `forM` and `secretShare`, for the bit vector and standard `secretShare` for the arithmetic value, resulting in `[Faceted ps '[] Bool]` and `Faceted ps '[] a` over the computing parties `ps`.

Using the ESBs, the conversion method from Arithmetic to Boolean Domain `A2B` proceeds as follows: First, the input value is masked and reconstructed:

$$t \leftarrow Declassify^A(\langle x \rangle^A - \langle r \rangle^A),$$

and this public value is then shared in the Boolean domain (B):

$$\langle t \rangle^B \leftarrow Share^B(t),$$

and after this, the mask is removed by performing a Boolean addition:

$$\langle x \rangle^B \leftarrow \langle t \rangle^B + \langle r \rangle^B.$$

On the other way around, to convert from Boolean to Arithmetic values, the `B2A` method proceeds similarly: We start with a set of ESBs and perform the following steps:

$$t \leftarrow Declassify^B(\langle x \rangle^B - \langle r \rangle^B).$$

Then, this public value is then shared in the Arithmetic domain (A)

$$\langle t \rangle^A \leftarrow Share^A(t),$$

and finally, the mask is then removed again with a Arithmetic addition

$$\langle x \rangle^A \leftarrow \langle t \rangle^A + \langle r \rangle^A.$$

In order to perform the Boolean addition and subtraction that was previously required, we used the gates outlined in [48], where circuits for addition and subtraction are constructed using 1-bit adders and 1-bit subtractors. A 1-bit adder calculates the carry-out and sum bits for addition in an efficient manner as follows: The carry-out bit is calculated as explained in Sharemind's carryBits function:

$$c_{i+1} = c_i \oplus ((x_i \oplus c_i) \wedge (y_i \oplus c_i)),$$

while the sum bit is

$$s_i = x_i \oplus y_i \oplus c_i$$

This construction, illustrated in Figure 3.24, uses four XOR gates and one AND gate, making it efficient. For subtraction, defined in two's complement as $x - y = x + \overline{y} + 1$, a 1-bit subtractor is used. Like the adder, this subtractor is computed efficiently with four XOR gates and one AND gate:

$$c_{i+1} = x_i \oplus ((x_i \oplus c_i) \wedge (y_i \oplus c_i))$$

and the subtraction bit is

$$d_i = x_i \oplus \overline{y}_i \oplus c_i,$$

which is illustrated in Figure 3.36.

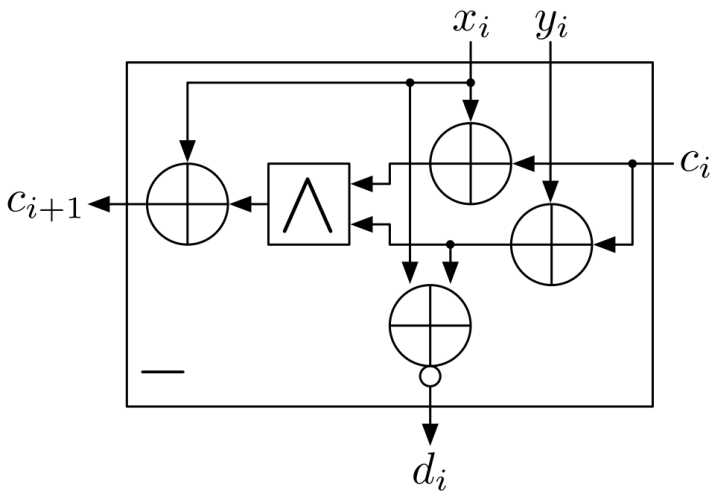

Fig 3.36 - 1-bit Subtractor (-) [48]

Remaining Operations Motion's paper lacks clear protocols for operations like division, equality checking, ordering, and multiplexing (`mux`). Consequently, we made use of our eDSL's default instantiations. These default instances naturally rely on previously defined Motion functions for addition and multiplication in both the arithmetic and Boolean domains, along with their respective conversion functions. Notably, division is still an open issue in Motion, and its integration into the framework remains an area for future work.

4. Analysis and Discussion of the Results

This chapter presents an evaluation of the proposed eDSL for secure multiparty computation, focusing on both its performance and its expressive power. The analysis is structured in three parts. First, we analyze how fundamental operations are executed in our Sharemind and Motion/Arithmetic-GMW backends in order to assess the performance at the level of individual primitives. In order to capture the overhead introduced by our approach, we perform a series of microbenchmarks that repeatedly execute atomic operations. We also study the scalability of GMW's primitives as the number of participants grows, since the Motion framework supports an arbitrary number of computing parties, a property we evaluate through our Motion/Arithmetic-GMW backend.

Secondly, we examine the performance of larger MPC programs that were originally implemented in Symphony and re-expressed in our eDSL. Both Sharemind and GMW are used to run these macrobenchmarks, and the results are examined with regard to the primitive costs provided in the microbenchmarks.

Lastly, we will examine the eDSL's expressiveness and usefulness. We compare its main features directly to Symphony, highlighting its support for delegation, resharing, and first-class party sets, as well as its capacity to implement demanding protocols like the LWZ secure shuffle, which were previously found to be outside the scope of the majority of MPC languages.

4.1. Performance Benchmarks

We begin with individual primitive microbenchmarks, comparing them across backends and analyzing GMW's scalability, before proceeding to whole-protocol executions.

4.1.1 Microbenchmarks

In this section, we evaluate and compare the primitives of the two MPC backends implemented in Chapter 3.3: Sharemind and Arithmetic-GMW. We consider the following operations: sharing, unsharing, resharing, multiplication, equality, ordering comparisons (greater-than-equal, less-than, less-than-equal, greater-than), and conversions between arithmetic and Boolean shares.

For comparison, we also include an insecure backend, where a single trusted party performs all computations directly, without any cryptographic protocol or inter-party communication. Using this baseline, we can calculate the overhead that the MPC protocols and infrastructure add in comparison to plain computation.

Comparison of Insecure, Sharemind, and Motion/GMW (3 Computing Parties) Given this setup, Table 4.1 reports the average execution time per operation (in seconds). Every operation was executed 100 times, and the execution time was recorded separately for each computing party using the Linux `time` command, specifically its user-time component, which measures CPU time spent in user mode. We started by calculating the average for each party over the course of 100 repetitions. Following that, the mean of these per-party averages was calculated, yielding the values shown in the table. The table distinguishes between "Sharemind," which uses its dedicated optimized protocols for equality and comparisons, and "Sharemind (Def)", where we forced it to use the generic defaults provided by our interface. GMW always relies on such generic defaults, as Motion does not expose specialized primitives for equality and ordering comparisons in our backend. This makes the table immediately reflect the distinction between optimized and generic constructions.

Tab 4.1 - Average execution time of basic operations across backends (in seconds)

	Share	Unshare	Reshare	Mult	Eq	Ge	Lt	Le	Gt	$A \to B$	$B \to A$
Insecure	0.0019	0.0019	0.0018	0.0017	0.0018	0.0019	0.0017	0.0019	0.0016	0.0017	0.0016
Sharemind	0.0026	0.0029	0.0026	0.0032	0.0156	0.0171	0.0175	0.0310	0.0308	0.0173	0.0082
Sharemind (Def)	-	-	0.0032	-	0.0296	0.0777	0.0779	0.0782	0.0774	-	-
GMW 3	0.0025	0.0030	0.0031	0.0036	0.0561	0.1339	0.1334	0.1358	0.1518	0.0322	0.0728

Figure 4.1 provides a visual comparison of these same results (in seconds) on a logarithmic scale, making the performance gap between Sharemind and GMW clearer.

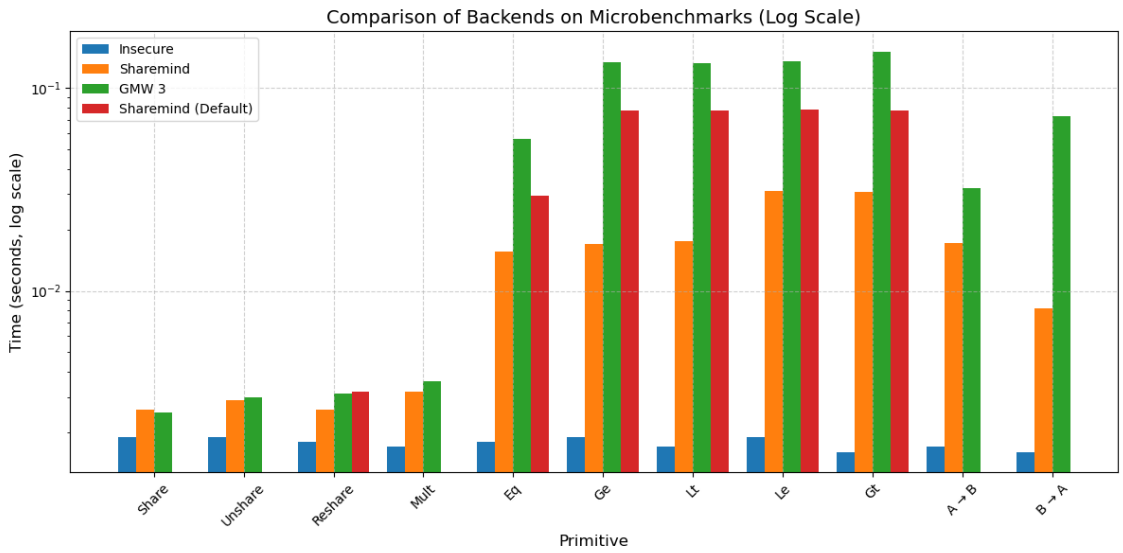

Fig 4.1 - Average execution time of basic operations across backends

The insecure baseline shows all primitives executing in the order of 1–2 milliseconds, and we suggest that this uniformity arises because, in this case, every operation reduces to a local pure computation and the cost is dominated by the runtime overhead rather than the primitive itself, making the differences between operations negligible. Sharing, Unsharing, and Resharing are also inexpensive across all backends (around 2-3 milliseconds), since both Sharemind and Arithmetic-GMW implement them in essentially the same way. The only distinction is that Sharemind offers a dedicated resharing protocol, whereas our GMW uses a generic default with a similar structure that generalizes to an arbitrary number of parties. Sharemind increases costs by roughly one order of magnitude for the remaining operations, with multiplications taking $0.0032s$ (≈ 2 times slower than insecure) and ordering operators (`Ge`, `Lt`, `Le`, `Gt`) around 0.015–$0.03s$ ($\approx$ 10–20 times slower). GMW with three computing parties, however, is still more expensive, particularly for comparison operators: equality requires 0.05-$0.06s$ (≈ 30 times slower than insecure, 3.5 times slower than Sharemind), while ordering operators require around $0.13 - 0.15s$ each. Multiplication remains relatively inexpensive at $0.0036s$, comparable to Sharemind. while conversion operations are also more costly in GMW, with $B \to A$ reaching $0.073s$.

These results show that while multiplication is still efficient on both backends, comparison operations account for the majority of GMW's costs. Our presumed existence of free Beaver triples helps to explain this since we did not model the offline phase required to generate these triples. In a real deployment, multiplication would therefore also be slower. The more prominent difference is in equality and comparisons, which in our implementation

expand into hundreds of additions and dozens of multiplications.

To confirm this interpretation, we also forced Sharemind to use its generic defaults (reported in Table 4.1 as “Sharemind (Def)”) rather than its specialized equality and comparison protocols. As shown in the table, this inflates costs by 2-4.5 times compared to “Sharemind,” which uses dedicated routines. This mirrors the pattern seen in GMW and suggests that generic constructions, not any fundamental inefficiency of GMW’s backend, are the main cause of the slowdown. This is also explicitly stated in Table 4.2, which demonstrates that Sharemind’s dedicated protocols require significantly fewer gates than default generic instances for equality and ordering.

Tab 4.2 - Gate counts per operation (ADD and MUL)

		Eq	Ge	Lt	Le	Gt	$A \to B$	$B \to A$
ADD gates	Sharemind	0	92	98	100	101	96	32
	Sharemind (Def)	161	353	352	353	352	-	-
	GMW 3	225	481	480	481	480	160	353
MUL gates	Sharemind	32	32	32	65	65	32	0
	Sharemind (Def)	65	192	192	192	192	-	-
	GMW 3	65	192	192	192	192	32	64

It is important to note that these metrics are not intended to give an exact assessment of the backends’ overall performance. The goal of the experimental setup was to record broad trends and orders of magnitude rather than to create fine-grained benchmarks. Therefore, rather than being conclusive comparisons of Sharemind and GMW, the differences we see emphasize the influence of optimized vs. generic constructions and our implementation choices. Nevertheless, the results consistently indicate that Sharemind achieves lower execution times than GMW across all considered primitives, and can therefore be regarded as the more efficient backend within the scope of our evaluation.

In addition to comparing Sharemind and GMW within our framework, it is useful to contextualize our results with the official benchmarks of these systems. For Sharemind, the original paper [39] reports both amortized vector costs and a “Single op.” column in milliseconds, which is directly comparable to our measurements. For instance, our evaluation yields 3.2 ms and 15.6 ms for multiplication and equality, respectively, but the publication reports 25.9 ms and 101 ms. Although our results are a little lower, the orders of magnitude

are consistent. This can be explained by the fact that Sharemind measurements include real network latencies, whereas in our system, each participant run as local processes communicating over HTTP on the loopback interface, eliminating network delays.

For GMW, the situation is different. The published benchmarks in [37] only report amortized costs over large SIMD batches (10^3-10^6 operations), in the nanosecond or microsecond range, and provide runtimes for primitive operations such as sharing, reconstruction (unsharing), and conversions, as well arithmetic operations, and comparisons. Notably, the paper reports equality and greater-than operations at approximately 0.1 milliseconds in a three-party local area network setting, but these figures are amortized over $1,000$ SIMD executions and are therefore not directly comparable to our single-operation benchmarks. Nevertheless, the discrepancy suggests that Motion includes optimized implementations of equality and comparison protocols, in the same spirit as Sharemind's specialized routines, although the paper does not describe them in detail. Therefore, rather than representing Motion's optimized capabilities, our results should be interpreted as conservative upper bounds that represent generic, backend-agnostic constructions.

Scalability in GMW As Motion is a N-party MPC framework, we studied how the cost of its primitives varies as the number of computing parties increases as shown in Table 4.3.

Tab 4.3 - Average GMW execution time with varying numbers of parties (in seconds)

	Share	Unshare	Reshare	Mult	Eq	Ge	Lt	Le	Gt	A $\rightarrow$ B	B $\rightarrow$ A
GMW 3	0.0025	0.0030	0.0031	0.0036	0.0561	0.1339	0.1334	0.1358	0.1518	0.0322	0.0728
GMW 5	0.0027	0.0033	0.0037	0.0039	0.0810	0.2124	0.2077	0.2096	0.2222	0.0412	0.1010
GMW 7	0.0027	0.0037	0.0046	0.0044	0.0997	0.2893	0.2984	0.2825	0.2813	0.0540	0.1365
GMW 9	0.0027	0.0040	0.0050	0.0050	0.1325	0.3724	0.3598	0.3627	0.3461	0.0646	0.1715

Sharing, unsharing, resharing, and multiplication all increase only slightly as the number of parties grows, but remain inexpensive overall, with costs stable in the range of 2–5 milliseconds. By contrast, equalities and comparisons expand significantly more steeply: equality increases from $0.056s$ to $0.133s$, and ordering operators almost triple in value, from roughly $0.13s$ to $0.36s$. Likewise, conversion operations scale significantly, with $A \rightarrow B$ and $B \rightarrow A$ nearly doubling their execution times, from $0.032s$ to $0.065s$ and from $0.073s$ to $0.172s$, respectively.

Figure 4.2 shows these scalability patterns.

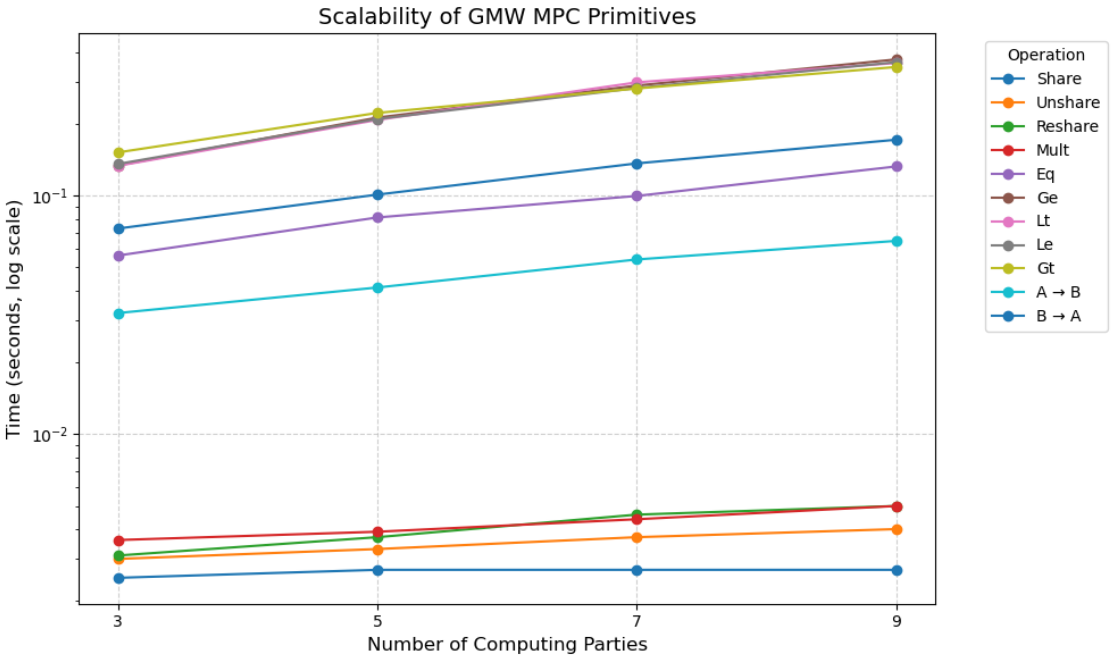

Fig 4.2 - Scalability of GMW primitives

Notably, we also measured how many addition and multiplication gates these primitives used for different numbers of computing parties. The counts persist the same as those shown in Table 4.2, demonstrating that the size of the census has no effect on the underlying circuit structure. Therefore, rather than a change in the cryptographic gate complexity of the protocols, the observed scalability trends in Table 4.3 are explained purely by the increase in communication costs among more parties.

Overall, the growth is close to linear in the number of parties, as expected given that these primitives in GMW require communication whose cost increases with the number of participants.

4.1.2 Macrobenchmarks

We now turn to the evaluation of full MPC programs that were re-implemented in our eDSL. These programs rely on the atomic primitives discussed in the previous section, allowing for a more realistic comparison of end-to-end performance between backends.

In our eDSL, we re-implemented a representative set of benchmarks that were previously articulated in Symphony [51]:

- **Hamming distance** - computes the Hamming distance between two arrays, as described in Chapter 3.2.2.

- **Edit distance** - dynamic programming algorithm for computing the edit distance between two arrays.
- **Private Set Intersection (PSI)** - intersection of two sets that are directly implemented using generic MPC primitives (referred to as naive PSI because there are more effective, specialized PSI protocols).
- **Richest** - an N-party variant of the Millionaires' problem.
- **LWZ shuffle** - secure shuffling of an array using N reshares of a linear secret sharing scheme.

In addition, we also implemented the **Crosstabulation** protocol from [52], which computes the sum of salaries per category given a salary table and a category table.

Given this, Table 4.4 reports the average execution time per computing party of these programs on the Insecure setting and the Sharemind, and GMW (3-party) backends.

Tab 4.4 - Average execution time of macrobenchmark programs (in seconds)

	Hamming	Edit	Cross	PSI	Richest	LWZ
Insecure	0.0016	0.0019	0.0018	0.0017	0.0016	0.0034
Sharemind	0.1334	0.8477	0.2177	0.1957	0.0833	0.0214
GMW 3	1.5791	7.8018	2.0205	0.8451	0.8286	0.0227

A visual comparison of these equivalent results on a logarithmic scale is shown in Figure 4.3, which highlights the performance difference between Sharemind and GMW, in seconds.

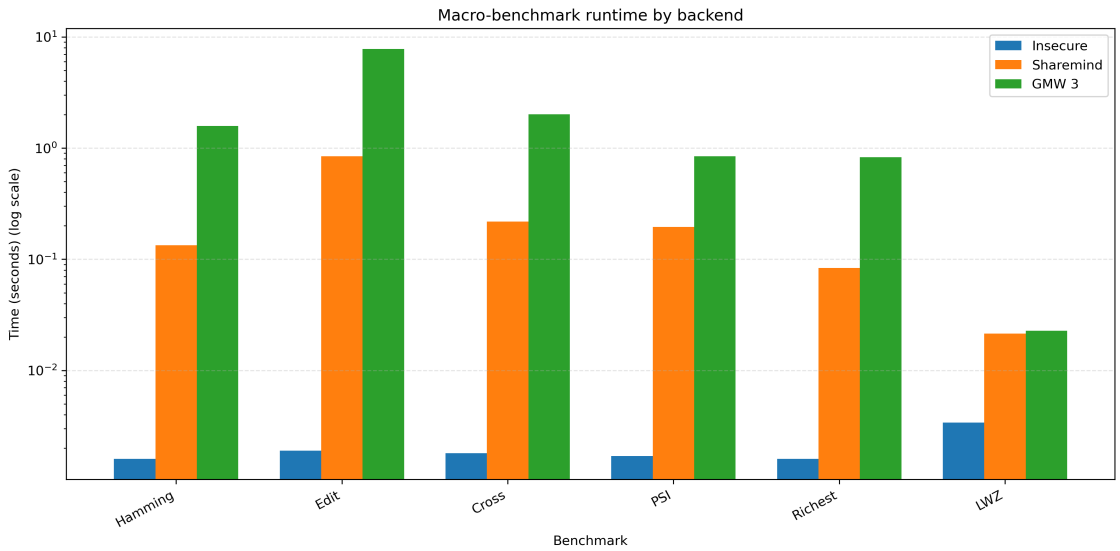

Fig 4.3 - Average execution time of macrobenchmark programs

The insecure baseline executes all programs in less than four milliseconds. We suggest that this uniformity again reflects that in the insecure setting, each program reduces to a local pure computation at a single party, and the measured time is dominated by the overhead of the choreographic and bootstrap runtime itself, such as initializing the computation and/or evaluating the monadic structure. As a result, the measurements are nearly constant regardless of the protocol's internal complexity, which explains why the execution times of individual primitives are very close to those of full MPC programs. Between the two MPC backends, Sharemind consistently outperforms GMW by roughly one order of magnitude across all programs except LWZ.

The LWZ shuffle is a special case since it relies solely on resharing between coalitions and randomized permutations rather than multiplication, equality, or comparison gates, which explains the minimal difference between backends and its low absolute cost. We return to LWZ in the next section, where it is used as a case study for expressiveness.

For the remaining programs, the differences directly reflect the primitive costs observed earlier. Hamming distance, Crosstabulation, PSI, and Richest rely on equality and conditional selection (`mux`). In addition to earlier gates, edit distance also involves repeated comparisons (`le`) inside a dynamic programming loop. Because our Motion/Arithmetic-GMW backend does not provide optimized equality and comparison operations, all of these programs run at considerably greater costs costs than Sharemind, roughly an order of magnitude slower.

It is important to note that our evaluation focuses on the backends built in ChoreoMPC rather than the original Symphony system. An obvious next step would be to compare the execution of the same macrobenchmarks, such as Hamming distance, Edit distance, etc, to Symphony's native implementation, highlighting differences in compiler optimizations, run-time infrastructure, and backend integration. However, such comparisons have drawbacks: Symphony and ChoreoMPC are written in distinct languages and use different execution models, which raises difficulties regarding how to measure performance variances. Evaluating programs within our framework allows for a more controlled setting to differentiate the costs of abstractions and backend choices. As a result, leave cross-system experiments to future work, pointing out that the current assessment provides a foundation for examining coordination, portability, and backend-specific performance considerations.

4.2. Expressiveness and Usefulness

In this section, we will go through the evaluation of the expressive power of our proposed eDSL, where we will compare it's main features with Symphony. As stated in Chapter 2.5.4, Symphony has become one of the most visible MPC languages in the literature because its design strongly emphasizes expressiveness, with the authors claiming that it can encode cryptographic constructions and coordination patterns not previously supported, including the LWZ secure shuffle protocol, which was initially outlined in [53]. Therefore, we directly compare our eDSL to Symphony in order to analyze its expressiveness and justify its design.

As previously demonstrated, Symphony builds upon previous linguistic approaches like Wysteria by introducing a number of innovative coordination and modularity constructs. Its primary characteristics include:

- **Scoped parallelism (`par` blocks):** which allows computations to be limited to arbitrary subsets of parties while assuring coordination;
- **First-class party sets:** where these are runtime values that can be dynamically constructed, passed as arguments, and used as execution guides;
- **Delegation:** which an input party can share its value with a compute coalition, allowing the computation to be carried out by other parties;
- **Resharing:** where shares can be transformed from one party set to another without decryption;

- **First-class shares:** where shares behave like regular values (they can be passed, stored, placed in data structures, delegated, and reshared).
- **Synchronized randomness:** the language offers effective primitives for reliably producing randomness among several parties.

These characteristics allow Symphony to express protocols like multi-phase auctions, committee elections, and the LWZ shuffle that call for dynamic coordination between shifting coalitions of parties.

We proceeded on to comparing the benefits of Symphony with what MultiChor has to offer. Although our eDSL does not adopt Symphony's surface syntax, the combination of MultiChor's choreographic constructs with our cryptographic interface provides equivalent expressive power, since we have support for:

- **Scoped Parallelism.** Symphony's par blocks allow only a subset of parties to execute a given expression. In ChoreoMPC, the same effect is achieved through MultiChor constructs such as `parallel`, `congruently`, and `enclave`, all of which restrict computation to a statically known party set. Thanks to census polymorphism and multiply-located values (`Located`, `Faceted`), we can parameterize computations over arbitrary subsets of parties, obtaining the same expressive power for controlling which coalitions execute which code.
- **First-class party Sets.** Symphony manipulates party sets at runtime. ChoreoMPC achieves an analogous form of expressiveness at the type level, via census polymorphism and subset proofs. Although party sets are resolved at compile time rather than at runtime, the practical effect is similar: our programs can be parameterized by arbitrary subsets, passed through functions, and used to generate combinatorial families of subsets (e.g., all committees of size t in LWZ). This gives us first-class manipulation of party sets, but realized through the Haskell type system rather than runtime data.
- **Delegation.** Symphony enables an input party to delegate computation to another coalition. In our eDSL, delegation emerges naturally from `share` and `unshare` in the MPC interface: an input party `P` shares a value to a compute set `Q`, who then carry out the MPC on its behalf. MultiChor's `parallel` and `enclave` constructs ensure that only `Q` participates in the cryptographic computation, mirroring the delegation semantics of Symphony.

- **Resharing.** Symphony treats resharing as a built-in primitive. In our eDSL, the `reshareTo` method of the `MPC` class expose the same operation. This operation is constructed using MultiChor's communication primitives, most notably `scatter`, together with `parallel` to perform local aggregation and type-directed subset reasoning. With specific implementations offered for both backends, the outcome replicates Symphony's semantics while staying backend-generic.

- **First-class shares.** In our system, `Secret protocol ps a` values play the role of Symphony's first-class shares. Since these are ordinary Haskell values, they can be freely embedded into tuples, lists, and higher-order functions, benefiting from the host language's abstraction mechanisms. This integration arguably surpasses Symphony's capabilities, because shares participate uniformly in Haskell's type class hierarchy, such as `SNum`, `SBool`, `SOrd` ... etc.

- **Synchronized randomness.** Symphony provides primitives for shared randomness. In our eDSL, this feature is exposed through the `syncRandAt` function in the interface, which is implemented using MultiChor's choreography operators to guarantee that each coalition member receives the same random value.

LWZ Shuffle

Symphony highlights the LWZ shuffle's expressiveness by pointing out that it requires repeated resharing values across coalitions that change dynamically. Previous MPC languages could not express this protocol without ad-hoc extensions. The LWZ protocol is a secure shuffle among a set of Q parties tolerating up to T corruptions. It proceeds in rounds where committees of size $|Q| - T$ receive shares of the input list, agree on a random permutation, permute their local shares accordingly, and then reshare the permuted list back to the whole group. By ensuring that any alignment of T corrupted parties is left out of at least one committee, the protocol guarantees the attacker cannot see the global shuffle.

In our system, the shuffle is implemented entirely within the generic MPC interface, using `reshareTo` from our `MPC` class for the resharing between committees, `syncRandAt` for synchronized randomness, and `SList` for list-based sharing and utilities. This allows the protocol to run on any backend that implements the required class instances. Unlike simpler protocols, which primarily rely on arithmetic gates, LWZ emphasizes coordination and resharing, and its successful implementation demonstrates that our interface achieves the same expressive ceiling as Symphony for secure computation.

However, Symphony's expressiveness remains confined to MPC-specific interaction patterns. Our eDSL, by contrast, inherits MultiChor's general-purpose choreographic foundations, where party sets are fully first-class and arbitrary communication between subsets of participants can be expressed without assuming a particular cryptographic model. This enables natural combinations of secure and insecure components, for instance, in our GMW backend the trusted generator acts as an ordinary party that locally produces Beaver triples (a non-cryptographic computation), yet is choreographically integrated into the secure multiplication protocol. It also supports general distributed applications that are unrelated to MPC, like the key-value store from [19] or the card game (Figure 2.5). These examples point out how our eDSL and MultiChor naturally expands to the larger universe of distributed choreographies, whereas Symphony is restricted to cryptographic protocols.

As explicitly stated in the MultiChor paper [16]:

> Programs in these languages [Wysteria and Symphony] are choreographies, and can exhibit census polymorphism, but both of these languages have homomorphic encryption baked into their semantics for communication, and they cannot be used for general-purpose choreographic programming

This shows that while we match Symphony within MPC, our eDSL exceeds it by supporting the broader space of distributed choreographies.

Table 4.5 summarizes the comparison above, placing Wysteria, Symphony, and our eDSL side by side.

Tab 4.5 - Expressiveness comparison: Wysteria [42], Symphony [38], and our eDSL

Feature	**Wysteria**	**Symphony**	**Our eDSL**
Scoped Parallelism	✓	✓ (par blocks)	✓ (`congruently/parallel`)
First-class party sets	✗	✓ (run-time values)	✓ (compile-time values)
Delegation	✗	✓	✓ (`share/unshare`)
Resharing	✗	✓	✓ (`reshareTo`)
First-class shares	✗	✓	✓ (ordinary Haskell values)
Synchronized randomness	✗	✓	✓ (`syncRandAt`)
General-purpose choreographies	✗	✗	✓ (arbitrary programs)

In addition, we also implemented two further Symphony programs, Analytics and Euclidean, which compute mean/variance over database joins and the minimum Euclidean distance

to a point, respectively. These programs rely on a secure division primitive, which is not yet available in our current Sharemind or GMW backends. For now, they are executed and tested in the insecure backend, but their implementation demonstrates that our framework can express such programs and will support benchmarking on additional backends as soon as the primitive is added.

Beyond expressiveness, our eDSL's Haskell embedding has additional benefits. Our protocols utilize common Haskell features, such as list processing, recursion, and higher-order functions, without requiring special control structures. They also inherit Haskell's larger ecosystem of functors, monads, and libraries. This leads to programs that are easier to comprehend than Symphony's standalone DSL. Furthermore, the design of our Interface around type classes (`SNum`, `SBool`, `SEq`, `SOrd`, etc.) makes protocols both datatype-polymorphic and backend-polymorphic: the same source code runs unchanged over the instantiated backends. This portability enables systematic benchmarking across frameworks and flexible deployment in practice, while the modular structure of the interface also makes the system easily extensible with new datatypes and operations. Additionally, given that badly written choreographies cannot even be expressed, MultiChor's type-level reasoning with subset proofs and Ghosts of Departed Proofs guarantees that participation sets get tracked statically, offering stronger guarantees than Symphony's runtime treatment of party sets.

5. Conclusions

In this thesis, we have investigated the integration of choreographic programming and secure multiparty computation, with the goal of combining the global reasoning and coordination guarantees of choreographies with the cryptographic security of MPC. This project resulted in ChoreoMPC, an embedded Haskell domain-specific language that allows for the design and implementation of MPC protocols within a choreographic model.

The creation of a unified backend-agnostic interface for MPC is the primary contribution of this work. The system proposes secure analogues of arithmetic, boolean, and ordering operations over secret-shared values through a hierarchy of type classes, such as `MPC`, `SNum`, `SBool`, etc. These abstractions are instantiated for a variety of backends, including Sharemind, Motion, and an insecure baseline, demonstrating that the same source code may be performed across many frameworks without modification. Backend portability is additionally improved by strong static assurances obtained by census polymorphism and type-level subset proofs made possible by MultiChor. Our findings show that the interface reaches Symphony's expressiveness for MPC, as demonstrated by our implementation of the LWZ shuffle within the generic interface, while supporting general-purpose choreographies beyond MPC, like the card game shown in Figure 2.5, through its integration with MultiChor. As a result, our language not only matches Symphony's expressiveness within MPC but also exceeds it overall by extending naturally to general distributed programs.

The implementation work demonstrates that choreographic principles naturally fit onto MPC protocols. Classical primitives like secret sharing, multiplication, and comparison were effectively re-expressed as choreographies, while more complex structures and algorithms like the Hamming distance and the LWZ shuffle were implemented within the generic interface. These case studies confirm the practicality of our approach, while also benefiting from the additional features that come with being incorporated into Haskell, like the ease of use of recursion, higher-order functions, list processing, and a larger ecosystem.

In general, this thesis has shown that choreographic programming provides clear benefits while also being compatible with secure multiparty computing. ChoreoMPC provides a high-level, type-safe, backend-polymorphic interface that simplifies the writing, execution,

and reasoning of MPC protocols. These findings help corroborate that choreographies can be adopted as a general paradigm for safe distributed computing.

5.1. Future Work

This thesis provides several directions for future study and improvement. At the protocol support level, important algorithms still require better implementation. Specifically, our current backend implementations of Sharemind and Motion do not yet support division protocols, which would increase the scope of numerical calculations that can be expressed in ChoreoMPC. For the generation of multiplication triples, our Motion backend might also profit from a more realistic offline phase, better separated from the online phase. This would improve benchmarks' realism and bring them more in line with MPC system best practices. Another area for improvement lies in designing more efficient equality and ordering protocols over words, since current implementations remain comparatively expensive.

The deployment of an actively secure backend is an additional research possibility. For simplicity, this thesis centered on passive adversaries, however, moving to an active model would only require replacing the underlying cryptographic subprotocols, not the high-level choreographies themselves. This implies that new backends could offer actively secure instances of the same programs, while ChoreoMPC's interface would remain stable.

Finally, it would be beneficial to run the original Symphony system and directly compare its performance with ChoreoMPC using the same protocols and inputs. This would give a more rigorous experimental foundation for comparing the relative benefits of the two approaches.

References

[1] F. Montesi, “Choreographic programming,” Ph.D. Thesis, IT University of Copenhagen, 2013, https://www.fabriziomontesi.com/files/choreographic-programming.pdf. [Cited on pages 1 and 6.]

[2] W3C, “Web services choreography description language version 1.0,” https://www.w3.org/TR/ws-cdl-10/, 2005, w3C Candidate Recommendation. [Cited on pages 3 and 4.]

[3] F. Montesi, *Introduction to Choreographies*. Cambridge University Press, 2023. [Cited on pages 3 and 12.]

[4] Z. Qiu, X. Zhao, C. Cai, and H. Yang, “Towards the theoretical foundation of choreography,” in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 973–982. [Online]. Available: https://doi.org/10.1145/1242572.1242704 [Cited on pages 4, 5, and 7.]

[5] M. Carbone, K. Honda, and N. Yoshida, “Structured communication-centered programming for web services,” *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 2, Jun. 2012. [Online]. Available: https://doi.org/10.1145/2220365.2220367 [Cited on pages 4 and 6.]

[6] F. Montesi. Choreographic programming. Accessed: 2025-07-13. [Online]. Available: https://www.fabriziomontesi.com/bliki/ChoreographicProgramming [Cited on pages xi and 5.]

[7] “Jolie programming language,” https://www.jolie-lang.org/, accessed: 2025-06-06. [Cited on page 7.]

[8] M. Carbone and F. Montesi, “Deadlock-freedom-by-design: multiparty asynchronous global programming,” in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 263–274. [Online]. Available: https://doi.org/10.1145/2429069.2429101 [Cited on page 7.]

[9] S. Giallorenzo, F. Montesi, and M. Peressotti, “Choral: Object-oriented choreographic programming,” *ACM Trans. Program. Lang. Syst.*, vol. 46, no. 1, Jan. 2024. [Online]. Available: https://doi.org/10.1145/3632398 [Cited on page 7.]

[10] L. Cruz-Filipe, E. Graversen, L. Lugović, F. Montesi, and M. Peressotti, “Functional choreographic programming,” *arXiv preprint arXiv:2111.03701*, 2021. [Online]. Available: https://arxiv.org/abs/2111.03701 [Cited on pages 8, 13, and 14.]

[11] A. K. Hirsch and D. Garg, “Pirouette: higher-order typed functional choreographies,” *Proc. ACM Program. Lang.*, vol. 6, no. POPL, Jan. 2022. [Online]. Available: https://doi.org/10.1145/3498684 [Cited on pages 8 and 13.]

[12] G. Shen, S. Kashiwa, and L. Kuper, “Haschor: Functional choreographic programming for all (functional pearl),” *Proc. ACM Program. Lang.*, vol. 7, no. ICFP, Aug. 2023. [Online]. Available: https://doi.org/10.1145/3607849 [Cited on pages xi, 9, 10, 11, and 12.]

[13] ——, “Haschor: Functional choreographic programming in haskell,” https://github.com/gshen42/HasChor, 2023, accessed: September 25, 2025. [Cited on pages xi and 10.]

[14] F. Montesi. (2023, May) Knowledge of choice. Last updated 2024-12-17. [Online]. Available: https://www.fabriziomontesi.com/bliki/KnowledgeOfChoice#CDP12 [Cited on page 12.]

[15] M. Bates, S. Kashiwa, S. Jafri, G. Shen, L. Kuper, and J. P. Near, “Efficient, portable, census-polymorphic choreographic programming,” *arXiv:2412.02107v2 [cs.PL]*, 2025. [Online]. Available: https://arxiv.org/abs/2412.02107v2 [Cited on page 12.]

[16] M. Bates, S. Jafri, and J. P. Near, “Multichor: Census polymorphic choreographic programming with multiply located values,” *arXiv:2406.13716 [cs.PL]*, 2024. [Online]. Available: https://arxiv.org/abs/2406.13716 [Cited on pages xi, 13, 14, 15, 18, 19, and 88.]

[17] M. Bates and J. P. Near, “We know i know you know; choreographic programming with multicast and multiply located values,” *arXiv preprint arXiv:2403.05417*, 2024, published: March 8, 2024. [Online]. Available: https://arxiv.org/abs/2403.05417 [Cited on pages 13 and 14.]

[18] M. Noonan, “Ghosts of departed proofs (functional pearl),” *SIGPLAN Not.*, vol. 53, no. 7, p. 119–131, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3299711.3242755 [Cited on page 18.]

[19] M. Bates, S. Jafri, and J. P. Near, “Multichor: Census polymorphic choreographic programming,” https://github.com/ShapeOfMatter/MultiChor, 2025, accessed: August 1, 2025. [Cited on pages xi, 18, 30, and 88.]

[20] ——, “Multichor: Census polymorphic choreographic programming,” https://github.com/ShapeOfMatter/MultiChor/blob/main/examples/CardGame.hs, 2025, accessed: September 25, 2025. [Cited on pages xi and 19.]

[21] D. Evans, V. Kolesnikov, and M. Rosulek, “A pragmatic introduction to secure multi-party computation,” *Found. Trends Priv. Secur.*, vol. 2, no. 2–3, p. 70–246, Dec. 2018. [Online]. Available: https://doi.org/10.1561/3300000019 [Cited on pages 20, 23, and 26.]

[22] A. Beimel and B. Chor, “Universally ideal secret sharing schemes (preliminary version),” in *Advances in Cryptology - CRYPTO ’92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, ser. Lecture Notes in Computer Science, vol. 740. Springer, 1992, pp. 183–195. [Cited on page 23.]

[23] A. C. Yao, “Protocols for secure computations,” in *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, ser. SFCS ’82. USA: IEEE Computer Society, 1982, p. 160–164. [Cited on pages 23 and 27.]

[24] O. Goldreich, S. Micali, and A. Wigderson, “How to play any mental game,” in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, ser. STOC ’87. New York, NY, USA: Association for Computing Machinery, 1987, p. 218–229. [Online]. Available: https://doi.org/10.1145/28395.28420 [Cited on pages 24, 27, and 28.]

[25] M. Ben-Or, S. Goldwasser, and A. Wigderson, “Completeness theorems for non-cryptographic fault-tolerant distributed computation,” in *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, ser. STOC ’88. New York, NY, USA: Association for Computing Machinery, 1988, p. 1–10. [Online]. Available: https://doi.org/10.1145/62212.62213 [Cited on page 24.]

[26] D. Chaum, C. Crépeau, and I. Damgard, “Multiparty unconditionally secure protocols,” in *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, ser. STOC ’88. New York, NY, USA: Association for Computing Machinery, 1988, p. 11–19. [Online]. Available: https://doi.org/10.1145/62212.62214 [Cited on page 24.]

[27] A. Jeong. Gmw: Input shares and communication. Accessed: 2025-07-07. [Online]. Available: https://blog.kroma.network/gmw-input-shares-and-communication-70803f4ae7ef [Cited on pages ix, xi, 24, and 25.]

[28] D. Beaver, “Efficient multiparty protocols using circuit randomization,” in *Advances in Cryptology — CRYPTO ’91*, J. Feigenbaum, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 420–432. [Cited on pages 26, 28, and 72.]

[29] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, “Fairplay—a secure two-party computation system,” in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM’04. USA: USENIX Association, 2004, p. 20. [Cited on page 27.]

[30] M. Geisler, “Cryptographic protocols:: Theory and implementation,” 2010. [Cited on page 27.]

[31] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos, “Sepia: privacy-preserving aggregation of multi-domain network events and statistics,” in *Proceedings of the 19th USENIX Conference on Security*, ser. USENIX Security’10. USA: USENIX Association, 2010, p. 15. [Cited on page 27.]

[32] X. Wang, A. J. Malozemoff, and J. Katz, “EMP-toolkit: Efficient multiparty computation toolkit,” 2016, accessed: 2025-09-15. [Online]. Available: https://github.com/emp-toolkit [Cited on page 27.]

[33] S. Zahur and D. Evans. (2015) Obliv-c: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Paper 2015/1153. [Online]. Available: https://eprint.iacr.org/2015/1153.pdf [Cited on page 27.]

[34] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, “Tinygarble: Highly compressed and scalable sequential garbled circuits,” in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 411–428. [Cited on page 27.]

[35] D. Bogdanov, S. Laur, and J. Willemson, “Sharemind: A framework for fast privacy-preserving computations,” in *Computer Security - ESORICS 2008*, S. Jajodia and

J. Lopez, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 192–206. [Cited on pages 27, 28, 57, and 63.]

[36] Y. Zhang, A. Steele, and M. Blanton, “Picco: a general-purpose compiler for private distributed computation,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 813–826. [Online]. Available: https://doi.org/10.1145/2508859.2516752 [Cited on page 27.]

[37] L. Braun, D. Demmler, T. Schneider, and O. Tkachenko, “Motion – a framework for mixed-protocol multi-party computation,” *ACM Trans. Priv. Secur.*, vol. 25, no. 2, Mar. 2022. [Online]. Available: https://doi.org/10.1145/3490390 [Cited on pages 27, 28, 72, and 81.]

[38] I. Sweet, D. Darais, D. Heath, W. Harris, R. Estes, and M. Hicks, “Symphony: Expressive secure multiparty computation with coordination,” *arXiv preprint arXiv:2302.10076*, Feb. 2023, available at https://arxiv.org/abs/2302.10076. [Cited on pages ix, 27, 29, 30, 33, and 88.]

[39] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson, “High-performance secure multiparty computation for data mining applications,” *Int. J. Inf. Secur.*, vol. 11, no. 6, p. 403–418, Nov. 2012. [Online]. Available: https://doi.org/10.1007/s10207-012-0177-2 [Cited on pages xi, 28, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, and 80.]

[40] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider, “Hycc: Compilation of hybrid protocols for practical secure computation,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 847–861. [Online]. Available: https://doi.org/10.1145/3243734.3243786 [Cited on page 28.]

[41] D. Beaver, S. Micali, and P. Rogaway, “The round complexity of secure protocols,” in *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, ser. STOC ’90. New York, NY, USA: Association for Computing Machinery, 1990, p. 503–513. [Online]. Available: https://doi.org/10.1145/100216.100287 [Cited on page 28.]

[42] A. Rastogi, M. A. Hammer, and M. Hicks, “Wysteria: A programming language for generic, mixed-mode multiparty computations,” in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 655–670. [Cited on pages ix, 29, and 88.]

[43] C. Acay, R. Recto, J. Gancher, A. C. Myers, and E. 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*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 740–755. [Online]. Available: https://doi.org/10.1145/3453483.3454074 [Cited on page 30.]

[44] C. Acay, J. Gancher, R. Recto, and A. C. Myers, “Secure synthesis of distributed cryptographic applications,” in *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*, 2024, pp. 433–448. [Cited on pages 31 and 33.]

[45] P. Laud and J. Randmets, “A domain-specific language for low-level secure multiparty computation protocols,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1492–1503. [Online]. Available: https://doi.org/10.1145/2810103.2813664 [Cited on pages 33, 39, 56, 57, and 63.]

[46] M. Greenberg, D. Gratzer, M. Sullivan, and D. Darais, “Symphony: A language for secure multiparty computation,” 2024, accessed: August 5, 2025. [Online]. Available: https://github.com/plum-umd/symphony-lang/blob/master/data/bin/Programming2023/benchmarks/hamming.sym [Cited on page 35.]

[47] C. Yan. Understanding hamming distance: A measure of similarity. Accessed: 2025-08-09. [Online]. Available: https://chrisyandata.medium.com/understanding-hamming-distance-a-measure-of-similarity-698ae2cb0ef6 [Cited on pages xi and 47.]

[48] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider, “Improved garbled circuit building blocks and applications to auctions and computing minima,” Cryptology ePrint Archive, Paper 2009/411, 2009. [Online]. Available: https://eprint.iacr.org/2009/411 [Cited on pages xi, 60, 61, 75, and 76.]

[49] P. Laud and A. Pankova, “Bit decomposition protocols in secure multiparty computation,” in *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC ’18. New York, NY, USA:

Association for Computing Machinery, 2018, p. 37–48. [Online]. Available: https://doi.org/10.1145/3267973.3267979 [Cited on pages xi, 61, and 62.]

[50] T. Granlund and P. L. Montgomery, “Division by invariant integers using multiplication,” *SIGPLAN Not.*, vol. 29, no. 6, p. 61–72, Jun. 1994. [Online]. Available: https://doi.org/10.1145/773473.178249 [Cited on page 67.]

[51] M. Greenberg, D. Gratzer, M. Sullivan, and D. Darais, “Symphony: A language for secure multiparty computation,” 2024, accessed: September 14, 2025. [Online]. Available: https://github.com/plum-umd/symphony-lang/tree/master/res/Programming23/symphony [Cited on page 82.]

[52] M. Keller, “How to scale multi-party computation,” in *Proceedings of the 2025 Australasian Computer Science Week*, ser. ACSW ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1–8. [Online]. Available: https://doi.org/10.1145/3727166.3727167 [Cited on page 83.]

[53] S. Laur, J. Willemson, and B. Zhang, “Round-efficient oblivious database manipulation,” in *Proceedings of the 14th International Conference on Information Security*, ser. ISC’11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 262–277. [Cited on page 85.]