Mechanizing and Implementing a Type System for Symphony

William Chung
The University of Maryland
Maryland, USA
wchung1@terpmail.umd.edu

ABSTRACT

Secure Multiparty Computation (MPC) is a cryptographic technique that allows multiple parties to compute a function jointly on their inputs, without revealing any information about them beyond what is learned from the output. Existing MPC languages are limited in their expressiveness, formal correctness, or ability to help developers coordinate parties. λ-SYMPHONY is a domain-specific language for MPC with an emphasis on coordination that extends lambda calculus. It is formalized with a draft of the syntax, semantics, and type system with a proof of soundness. A dynamically typed interpreter for Symphony implements and extends λ-SYMPHONY's semantics.

I mechanize λ-SYMPHONY's type system in the Coq proof assistant, thereby increasing confidence in its proof of soundness. Likewise, I implement the type system as an extension to the existing Symphony interpreter. Both implementations alter the original formalization. I find that useful MPC programs written by an expressive language with party coordination features can be statically checked based on a formal type system.

1 INTRODUCTION

Secure Multiparty Computation (MPC) gives multiple parties the ability to compute a joint function over their private data in a way where the execution of the function reveals nothing more about their data than what the output shows. The value of this field leads to the creation of different programming languages that can translate a user-defined, general-purpose program into a secure computation protocol.

Unfortunately, many programming languages do not have a formal static type system. Hastings et al. [3] observe in their SoK that these programming languages have correctness issues and silent failures due to their lack of type rules. In addition, some of these languages fail to provide a way for users to coordinate parties from being in the scope, limiting the language's practicality and efficiency. They also claim that even existing languages with static type systems and coordination features have issues with expressiveness. An elaboration of the issues with existing languages is provided in section 2.

There should be a formal type system and static type checker for an expressive MPC language that handles party coordination. Due to the problems of existing static and non-static MPC languages, I approach Symphony, an expressive domain-specific language with a focus on party coordination. Symphony resolves the problems of existing MPC languages by having an interpreter with many features to help developers write relevant MPC programs and an associated formal core calculus, called λ-SYMPHONY, with a type system. However, Symphony also has its own concerns as λ-SYMPHONY's type system has an unclear proof of soundness with minor bugs and is not implemented in the expressive Symphony interpreter. Addressing these concerns, I mechanize λ-SYMPHONY in the Coq proof assistant [8] to build confidence in its proof of soundness and implement the type system in the Symphony interpreter as a static type checker.

2 RELATED WORK

Existing MPC frameworks provide abstractions to help develop secure computation protocols. Some of them offer many expressive features but do not include a formal type system and avoid confronting the issue of party coordination. Other languages that are based on party coordination have their own challenges.

2.1 General MPC languages

Many languages extend a language to include expressive and familiar features. One of them includes Obliv-C which extends C. Being an extension of C, it includes well-known features such as recursive calls and pointers [9]. Another language is OblivVM which extends Java [4]. It is one of the first MPC languages to add native primitives, random types, and generic constants. These languages are among those compared in the SoK of Hastings et al. [3].

However, while these languages are expressive, Hastings et al. [3] observe that many of the languages had correctness issues, some not being reported to the user. They recommend taking a more principled approach to language design and verification based on programming language research to reduce these issues. Specifically, they suggest designing and implementing type rules to prevent the possibility of an incorrect or unexpected outcome.

2.2 Wysteria

Wysteria is a functional programming language with a refinement type system that provides support for mixed-mode programs [6]. It handles party coordination through two language constructs called a parallel block and secure block, leading to programs that can combine both local and private computations. The language comes with a formal core calculus, λ_{wy}, that establishes the foundations behind Wysteria's semantics and type system. The creators prove type soundness of the formal core calculus which leads to the demonstration that all inter-principal communication and information leakage occur in secure blocks.

Despite Wysteria's main feature being its secure block construct, the body inside this construct has limitations for common programming language features such as recursion. These limitations lead to trouble when I tried to write the greatest common denominator function in Wysteria. In addition, Hastings et al. [3] find general constraints such as a lack of support for divisions by numbers other than two and a lack of logical operators for Booleans. These restrictions hinder users from being able to write expressive programs for MPC.

2.3 λ-SYMPHONY

λ-SYMPHONY is an expressive and concise domain-specific language for MPC [1]. It extends lambda calculus in a way that helps the developer know which parties have access to which values, control which parties compute what, and mix MPC with cleartext computation. Specifically, its introduction of the parallel expression (par block) allows developers to control which parties would do the computation for a part of the program. The set of parties that compute is called the mode. λ-SYMPHONY is more expressive than Wysteria, providing first-class support for shares leading to more flexibility in using them. Darais et al. [1] formalizes λ-SYMPHONY, providing syntax, single-threaded semantics, type system, and distributed semantics with a proof of soundness.

While the formalization proves progress and preservation for type soundness, the proofs are unclear and skip over some details, leading to an opportunity for future elaboration. Like Wysteria, λ-SYMPHONY has an interpreter that extends the formal core calculus. However, unlike the formal core calculus it is based on, the Symphony interpreter is dynamically checked as it does not have a static type checker. Due to not being statically typed, the interpreter cannot be ensured to never get stuck.

3 MOTIVATION AND APPROACH

Having observed that previous approaches have limits in either expressiveness, correctness, or party coordination, I choose to take an already expressive MPC language, formalize it with an associated type system, and implement a type checker based on it. A language that fits this approach is the Symphony interpreter due to its expressiveness and associated formal language, λ-SYMPHONY. While the Symphony interpreter described by Sweet et al. [7] is not statically typed, the type system of λ-SYMPHONY can be extended and used to type check the interpreter.

The proofs in the current λ-SYMPHONY paper have problems in completeness, including a minor bug. To fix these issues, I reproduce and mechanize the proof of soundness for λ-SYMPHONY in the Coq proof assistant. The nature of Coq being a proof checker means that I can say λ-SYMPHONY 's type system is safe with greater confidence.

I wrote the prototype type checker for the Symphony interpreter in Haskell and helped integrate it as a command for the Symphony interpreter to detect static type errors.

4 MECHANIZATION OF λ-SYMPHONY.

The components of the mechanization include the following four parts from the λ-SYMPHONY formalization by Darais et al. [1]:

- Syntax: The syntax, from figure 5 of the formalization, can be represented with inductive datatypes that represent terms. These will include all expressions and atomic expressions.
- Operational Semantics: The small-step single-threaded semantics, from figure 7 of the formalization, can be represented by inductive propositions that take a term and return a proposition based off it.
- Type System: Types, from figure 5 of the formalization, can be represented by an inductive datatype. The rules of the typing system in figure 8 of the formalization can be implemented through an inductive proposition that takes a term and type and returns a proposition that type checks it.
- Proof of Soundness: A proposition that shows that a configuration in λ-SYMPHONY will always be multistep to a value. This can be shown by making helper theorems in Coq to prove progress and preservation as defined in theorem 5.2 and 5.3 of the formalization.

The following key features of λ-SYMPHONY are part of the mechanization:

- Normal language features: Base literals, binary operations, atomic conditionals, product types, sum types, IO operations, let expressions, function calls
  - Some features from the paper such as references, subtyping, and recursive types are admitted due to time restraints
- Par blocks and MPC: Parallel execution, share expressions, embedded expressions, and reveal expressions
  - These four features are vital to Symphony and reproducing proofs that include these will improve confidence in the usefulness of the typing system

4.1 Implementation Notes

4.1.1 Syntax and Semantics. Inductive datatypes mainly represent the context free grammar of the syntax from figure 5. These inductive data types include representations of base literals, protocols, base types, types, binary operations, expressions, atomic expressions, and types for binary operators. The atomic and normal expressions include both basic and MPC features. Definitions using modules in Coq represent other non-terminals in the syntax. The modules include the String module to help represent variables and the ListSet module to help represent sets of parties. Also, the ZArith_base module helps represent integer literals when they are needed.

Inductive datatypes represent the semantics metafunctions in figure 6 and semantics in figure 7 of the formalization. Inductive datatypes also represent values, the stack, and a configuration. I use the map module given in my program analysis course to make an environment. In addition, Inductive propositions represent the rules in the small-step operational semantics in figure 7 and the rule to determine a well-formed protocol in figure 9 that would be used in the semantics. A fixpoint and equivalent proposition represent the slice function, from the operational semantics, since the fixpoint guarantee one result while the proposition is easier to use for inductive propositions.

4.1.2 Type System. I implement the context as a partial map of types. I also implement well-formedness (type compatibility) and typing rules of the type system in figure 8 and figure 9 through inductive proposition with different cases for each typing rule.

4.1.3 Proof of Soundness. In this mechanization, I prove soundness of the typing system by proving the theorem of progress and preservation listed in Theorem 5.2 and 5.3 of the formalization. I prove progress of a configuration by first defining what a terminal configuration is based on Theorem 5.1. Then, I prove many supporting lemmas such as canonical lemmas for each value, lemmas

Figure 1: Proving the Type of Millionaire's Problem

```
Definition xexpr: expr := ParE ["A"] (AtomicE ReadIntE).

Definition yexpr: expr := ParE ["B"] (AtomicE ReadIntE).

Definition sxexpr: expr := (ParE ["A"; "C"] (AtomicE (ShareE ["A"] [ "C"]  "x"))).

Definition syexpr: expr := (ParE ["B"; "C"] (AtomicE (ShareE ["B"] ["C" ] "y"))).

Definition rexpr: expr :=(ParE ["C"] (AtomicE (BinopE LTOpp "sx" "sy"))).

Definition zexpr: expr := (ParE ["A"; "C"] (AtomicE (RevealE ["A"] "r"))).

Definition millExpr: expr := (ParE ["A"; "B"; "C"]
    (LetE "x" xexpr (LetE "y" yexpr (LetE "sx" sxexpr
    (LetE "sy" syexpr (LetE "r" rexpr (LetE "z" zexpr (ParE ["A"] (AtomicE (WriteE "z"))))))))))).

Definition all := ["A"; "B"; "C"].

Theorem testMillionare: (has_type_expr empty_context all millExpr (LocBaseTy (BoolTy) (ClearText) ["A"]) ).

  apply T_Par with ["A"; "B"; "C"]. reflexivity.
  (* Proves the type of x *)
  + apply T_Let with (LocBaseTy (IntTy) (ClearText) ["A"]). apply T_Par with ["A"].
  reflexivity. apply T_ReadInt with "A".  reflexivity. intros H. inversion H.
  (* Proves the type of y *)
  -  apply T_Let with (LocBaseTy (IntTy) (ClearText) ["B"]). apply T_Par with ["B"].
  reflexivity. apply T_ReadInt with "B".  reflexivity. intros H. inversion H.
  (* Proves type of sx *)
  *  apply T_Let with (LocBaseTy (IntTy) (Enc ["C"]) ["C"]). apply T_Par with [ "A"; "C"].
  reflexivity; auto. apply T_Share with "A" ["A"]; auto.  apply T_Var. apply WF_Base.
  split; reflexivity. reflexivity. intros H. inversion H. reflexivity. split; reflexivity.
  intros H. inversion H.
  (* Proves type of sy *)
  ** apply T_Let with (LocBaseTy (IntTy) (Enc [ "C"]) [ "C"]). apply T_Par with ["B"; "C"].
  reflexivity; auto. apply T_Share with "B" ["B"]; auto.  apply T_Var. apply WF_Base.
  split; reflexivity. reflexivity. intros H. inversion H. reflexivity. split; reflexivity.
  intros H. inversion H.
  (* Proves type of r *)
  ***  apply T_Let with (LocBaseTy (BoolTy) (Enc [ "C"]) ["C"]). apply T_Par with [ "C"].
  reflexivity. apply T_Binop with IntTy IntTy;  auto.  apply T_Var.  apply WF_Base.
  split; reflexivity. reflexivity. apply T_Var.  apply WF_Base. split; reflexivity.
  reflexivity. intros H. inversion H.
  (* Proves type of z *)
  **** apply T_Let with (LocBaseTy (BoolTy) (ClearText) [ "A"] ). apply T_Par with ["A"; "C"].
  reflexivity. apply T_Reveal with ["C"].  apply T_Var.  apply WF_Base.
  split; reflexivity.  reflexivity. intros H. inversion H. intros H. inversion H.
  unfold is_union. assert (party_set_union ["C"] ["A"] = party_set_union ["A"] ["C"]).
  apply party_set_union_sym. rewrite H. reflexivity. unfold not. intros H. inversion H.
  (*  Proves the type of body expression and does a check on the most outer par block's party set *)
  (* to prove the type of the whole expression *)
  apply T_Par with ["A"]. reflexivity.  apply T_Write with "A". apply T_Var.
  apply WF_Base. split; reflexivity. reflexivity. reflexivity. intros H. inversion H.
  + intros H. inversion H.
  Qed.
```

that show what happens when a variable expression is well-typed, and that slicing a variable expression preserves its type based on Lemma B.2 and B.3 of the formalization. I use these lemmas to prove a modified version of Lemma 5.7 that atomic expressions have the property of progress. Proving this lemma ranged from applying the typing rule, such as for the T-Base case, to having to apply one to all the existing lemmas to show that a certain value was desired for the small step semantics rule, such as for the T-Binop case. To prove progress of a configuration, I do case analysis and apply the progress of atomic expressions for some cases, and use the existing helper lemmas that prove progress of atomic expressions in some of the other cases.

I prove preservation of a configuration by proving preservation of atomic expressions similar to Lemma B.9 of the formalization. This proof uses some of the previous helper lemmas described above many times, some automation, and much injection. More helper lemmas I defined myself include lemmas to help show how a configuration could be well-typed such as how an environment could be well-formed depending after being properly updated or a type could always be well-formed if it is well-formed when the party set is empty. By using the environment update and the variable slice preservation lemma many times and Lemma B.9, I prove the preservation property of configurations by using inversion to go through each typing rule and operational semantic rule.

I then define a stuck configuration as a configuration being not terminal and being unable to take a multistep. After, I apply my proofs of progress and preservation with induction to prove soundness for configurations, the property that a well-typed configuration can never reach a stuck state.

4.2 Differences from the Original Formalization

4.2.1 Syntax and Semantics. Base literals which mean integers are not the only base type affecting various inductive data and propositions. In addition, located types and located values are both respectively flattened in the type and value inductive datatypes. I make this decision to avoid the complexity of proving equivalent propositions for two mutually inductive datatypes in Coq.

4.2.2 Type System. I implemented most of the typing rules inductive proposition in Coq similarly to its original formalization. Most changes in the type systems are based on the changes to base literals and the flattening of located values and types. Two new changes are made to prove type soundness. The mechanization requires that no element in the stack can be the empty party set if the stack is well-typed which is vital to proving progress. The other change is adding that party set p cannot be empty in T-Reveal to make later proofs in Coq easier.

4.2.3 Proof of Soundness. The original formalization includes simpler proofs for progress and preservation than what Coq requires as the paper skips over some details. Due to this fact, while I base the theorems of progress and preservation on the ones in the paper, I use different lemmas and different methods to reach the same end goal.

4.3 Results

Due to the Coq program of the mechanization being well-typed, the proof is logically sound enforcing that the essential parts of the type system in λ-SYMPHONY are type-safe.

In addition, the following proposition and proof in Coq in Figure 1 demonstrate an example of the mechanization in action. The expression showcases the MPC features specific to λ-SYMPHONY in a variation of the Millionaire's problem. The expression is a simpler version of figure 3 in the formalization. It has two parties input their net worth, one party computes which net worth is higher, and one of the input parties outputs which net worth is higher. The expression being constructible and well-typed displays how the mechanized language can allow two parties to reveal who has the higher net worth to a third party without revealing anything else. The Coq proof uses typing rules from the mechanization's type checking inductive datatype and various tactics to derive that the expression does have the given type showing successful use of the mechanization.

Lastly, implementing the mechanization reveals a minor bug in the original proof of soundness. A proposition that was originally not in the paper that was needed for a Coq proof to be correct was that a well-typed stack could contain an empty party set. Overall, the Coq mechanization is able to reduce bugs from the original paper which was able to minimize ambiguity from the original type system.

4.4 Potential improvements

In the future, this mechanization can add references, recursive types, and subtyping. Improvements to the mechanization also include less repetitive proofs and clearer variable names. In addition, the mechanization should use a more, accurate module to represent sets with more expressive lemmas rather than ListSet.

5 TYPE CHECKER IMPLEMENTATION

I implement the type checker for the Symphony interpreter in Haskell. Furthermore, I make use of The University of Vermont's version of the Haskell standard library. The parsed abstract syntax tree given to the interpreter is the input for the type checker.

Some type rules are simple to implement. For others, I use techniques based on type checker implementations shown by a book on type systems by Pierce [5], such as the use of joins, to turn the declarative rules defined in both λ-SYMPHONY and the Coq mechanization into algorithms. Moreover, since the Symphony interpreter both generalizes and extends λ-SYMPHONY, the type checker generalizes and extends the type system for the type checker. An example is that certain expressions are now able to contain expressions other than variables.

I design the type checker to accommodate both λ-SYMPHONY 's type system and interpreter.

5.1 Judgment

The judgment of the expression consists of the current mode, typing context, mode scope context, the set of the pre-defined principal literals, and a source context. A structure with associated expression monads represents the judgment itself. A constructor that is either the set of all principals or a power set of principal values represents

the current mode. It is also wrapped around by another constructor to handle the first-class principal set extension. An environment that maps variables to types represents the typing context. A dictionary that maps variables to modes represents the mode scope context. This mode scope context keeps track of variables bound by polymorphic lambdas and will be used in checks of type compatibility. A set of variables represents the set of pre-defined principal literals to keep track of which principal expressions are literals or variables. The source context is a structure containing details of the expression relative to the program such as line number. These parameters in the judgment structure are updated through a monad in rules such as T-Let for the typing context and accessed through the monad in rules such as T-Var.

5.2 Type Checking and Type Synthesis

The type checker is bidirectional, using a mix of functions that do type checking and type synthesizing.

I implement typing rules where the type generated is more ambiguous with a type-checking function. An example is T-Fun where the type of the lambda is added in the type context in the premise to show that a function is that type. In this case, checking that an expression is of a given type is easier than generating it. These functions return a monad of the unit to indicate whether or not it could be checked without an error or the error. Other expressions that are forced to be checked with a given type include sum injections, the nill expression, and the fold expression.

Meanwhile, rules such as T-Int that introduce the type, are relatively simple rules to implement as a type synthesis function. To elaborate, the located type $\sigma@m$, for a type σ depending on the rule, can be synthesized by accessing and converting the current mode m from the expression monad. These synthesis functions return a monad of a type it generates or an error.

In addition to deciding when to implement a typing rule as type checking or synthesis, I also determine when to switch between checking and synthesis for a rule. An example of using a checking function within synthesizing function is to determine if it is was appropriate to return the type in the annotation for an annotated expression. An example of using a synthesis function within a checking function is to synthesize the type of an expression to compare this type with a given type.

In general, a paper on bidirectional typing by Dunfield and Krishnaswami [2] gives much guidance to determine when it was appropriate to implement a rule as a type checking or type synthesis function.

5.3 Subtyping

Instead of a general rule for subsumption, λ-SYMPHONY originally has most of its expressions only containing variables instead of other sub-expressions and has T-Var put a subtyping check in the premise. This method makes it easier to implement premises that require types to be synthesized to a certain type in the type checker. While these expressions can now contain any sub-expression in the interpreter, a function that determines whether it is a subtype of a requested type can be called to fulfill the premise during its type check. The subtyping rules in the paper influence this function. The function takes two types and a set of subtyping assumptions for type variables as its arguments.

There are also times when the types to return were ambiguous due to many types being a supertype of the same types such as in T-If. To alleviate this problem, the type checker uses a technique in the book on type systems of using a join function to return the optimal supertype. I implement the join function based on the subtyping rules in λ-SYMPHONY. The located type rule suggests that join of located types is the inverse of joining party sets.

The type checker also uses the subtyping function in situations that check if expressions could be synthesized to a certain type due to how subsumption works.

5.4 Type Compatibility

The implemented well-formedness function both checks that a type matches the syntax from the paper and that the type is type compatible in the mode scope and current mode as defined by the rules in figure 9 of the formalization by Darais et al. [1]. In addition, the type checker enforces that any type returned from the general synthesizing function is well-formed based on the current mode and mode scope. An instance of this enforcement is checking the type given in a type annotation is well-formed using the self-described function.

In addition, the type check emulates the type compatibility check of the supertype synthesized in T-Var with a function implemented that returns the optimal supertype that is type compatible with the current mode and mode scope. For a located type inputted, the located type outputted has its location as the intersection of its existing location and the current mode.

5.5 Extensions to Data Types

Various extensions, additions, and changes are made from λ-SYMPHONY to the symphony interpreter that causes the type checker to extend the original type system. Pairs can be located which motivates me to implement their typing rules in the type checker similar to other located types' typing rules. The interpreter generalizes basic primitives, so the type checker generalizes their associated rules. Extensions include lists, which are implemented similar to pairs, and arrays, which are implemented similar to references. Also, the type checker extends case expressions to be more like pattern matching and adds if expressions as a special case for case expressions where the guard has to be a Boolean. Due to the language being statically typed, I implement recursive types and polymorphic types in the type checker based on the original typing rules in the formalization and book on type systems. I also implement a change to the original subtyping rule for recursive types in that when checking if the body is a subtype of the other, the type variable is assumed to be a subtype of the other, making the rule sound.

A newly added feature to the interpreter is implicit wire bundles discussed in the original draft. These values make different principals have their own, local version of a value. I implement their associated synthesis functions in the type checker to be consistent with the bundle's syntax and semantics in the interpreter.

5.6 Par blocks and MPC

I implement T-Par, one of the rules for type checking parallel expressions, in the type checker by modifying the mode to the non-empty intersection of the current mode and given mode using a monad when synthesizing the body expression. I implement T-ParEmpty, the other typing rule when the intersection of the current mode and given mode is empty, by synthesizing a type that is well-formed for the empty mode or checking that the given type is type compatible with the empty mode.

The other MPC rule implementations also make use of modifying the mode in the judgement and using party set operations. Moreover, for share and reveal expressions, their associate synthesis functions take a type without locations and make cleartext and encrypted versions with locations for future use. The type checker generalizes these implementations to accept more types such as all primitive types, sums, pairs, lists, and arrays in the type checker implementation.

A new MPC expression called send, originating from the interpreter, has a similar typing rule implementation to share, but instead returns a cleartext type as opposed to an encrypted type.

The last change that is accounted for is that cleartext types are implicitly embedded in the interpreter instead of explicitly using an embed expression. To account that a cleartext expression can be implicitly transformed into an encrypted expression, the subtyping function contains the case that cleartext types are subtypes of their encrypted type counterpart.

5.7 MPC Operations

A change accounted for when implementing the MPC operations of primitive operations, mux if, and mux case is that there can be expressions of cleartext types within encrypted typed expressions as the cleartext typed expressions would be embedded. The type checker also generalizes operations to take expressions of any type that can be shared due to the interpreter generalizing them.

5.8 First-class party sets

The interpreter can use party sets as data, while λ-SYMPHONY could not. I implement their introduction rules straightforwardly with the type checker accessing the typing context. However, I put more thought into how to handle set operations on principal sets that contain variables or are themselves variables since the results cannot be determined till runtime. To alleviate this issue, I add a constructor to the mode datatype to represent if the mode could not be determined till runtime or not. If it could not be determined meaning it can be any party set, any operation that takes it as an argument returns true if it needs to return a boolean or the mode that represents any mode if it needs to return a mode. As a result, this implementation allows programs that use statically non-deterministic principal sets to be statically checked for other potential type errors.

5.9 Top Level

At the top level, the principals, declarations, and definitions are given. The type checker adds the principals to the typing context with the appropriate type and to the set of principal literals. The type checker design handles declarations in a way where function declarations can be well-formed with any mode while other declarations can be well-formed with the mode of any principal. This check makes it so function types can be type compatible in another mode other than the mode of all principal sets which gives the developer more flexibility on when to call a function. The definition expressions are then checked with either the type checking functions that handle lambdas or general expressions depending on how it was previously declared. In general, the type checker returns a monad of the type of what the main function returns or the type error that results from it.

```
principal A B C D E

-- gmw = semi-honest, N-party, boolean sharing

def main : (unit@all -> bool@{E})@all
def main () = par {A,B,C,D,E}
  let a = par {A,C,D} share [gmw, bool : {A} → {C,D}]
  (par {A} read bool from "delegation.txt") in
  let b = par {B,C,D} share [gmw, bool : {B} → {C,D}]
  (par {B} read bool from "delegation.txt") in
  let c = par {C,D} a && b in
  par {C,D,E} reveal [gmw, bool : {C,D} → {E}] c
```

Figure 2: Symphony Delegations Benchmark Program

6 TYPE CHECKER RESULTS

Overall, I implement all of the features in λ-SYMPHONY and the essential features of the Symphony interpreter in the type checker. The type checker checks a Symphony program through an integrated Symphony command. I also implement around 30 basic example programs to test the typing rules in the type checker. All the source code and test programs can be found in the GitHub repository at https://github.com/plum-umd/symphony-lang/tree/types-rebase.

6.1 Benchmark Programs

To make sure relevant programs could be type checked, the type checker has been tested on typed annotated versions of various programs of the benchmark suite that tested the interpreter. One of these benchmarks is the Delegations program in figure 2. This program takes the Boolean decisions of two input parties. Two computing parties see if they both are true. They then reveal the result to an outputting party called E. The type checker can successfully show that the main of this program returned a Boolean of location E.

Figure 3 shows an example of a variation of the delegations program that fails to type check due to having the wrong party set as the revealing mode in the reveal expression. The type checker returns an error when given this program to showcase the issue that it was expecting a variable c to be of a subtype to be consistent with the mode and type given in the share expression.

```
principal A B C D E

-- gmw = semi-honest, N-party, boolean sharing

def main : (unit@all -> bool@{E})@all
def main () = par {A,B,C,D,E}
  let a = par {A,C,D} share [gmw, bool : {A} → {C,D}]
  (par {A} read bool from "delegation.txt") in
  let b = par {B,C,D} share [gmw, bool : {B} → {C,D}]
  (par {B} read bool from "delegation.txt") in
  let c = par {C,D} a && b in
  par {A,B,E} reveal [gmw, bool : {A,B} → {E}] c
```

Figure 3: Erroneous Symphony Delegations Benchmark Program

Lastly, figure 4 showcases a more complicated program that finds the Hamming distance of two strings inputted by party A and party B and reveals it to party A without leaking any private data. The type checker can type check this program as a natural number with location of party A. The output for these benchmarks can be shown by running them with the type checker as found in the GitHub repository. These benchmarks showcase that the type checker can successfully check programs that perform relevant multi-party computations.

6.2 Future Work

In the future, we can find a better way to handle type checking programs with variable party sets. One of these ways includes implementing refinement types to get more information about these variable party sets. The type system could draw some influence from Wysteria's refinement type system as it also has first-class principal sets [6].

7 CONCLUSION

Expressive, statically typed MPC languages that can help a developer coordinate parties are beneficial for helping developers make correct MPC protocols. Existing MPC languages have issues addressing the concerns of expressiveness, formal correctness, and party coordination. Learning from their issues, I implement a type checker for the expressive Symphony interpreter based on λ-SYMPHONY's type system assured by a mechanization in the Coq proof assistant. Both the mechanization's and type checker's designs accommodate adjustments from the original λ-SYMPHONY's type system. As a result, the type checker can detect type errors in relevant MPC programs written in Symphony.

In the future, the mechanization can strengthen the confidence of the type system more by adding more features from the type checker implemented. Meanwhile, the type checker can implement refinement types to handle variable party sets better.

REFERENCES

[1] David Darais, David Heath, Ryan Estes, William Harris, and Michael Hicks. 2020. λ-Symphony: A concise language model for MPC.

[2] Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (may 2021), 38 pages. https://doi.org/10.1145/3450952


```
principal A B

def hammingDist : ( (array{{A, B}}((nat[gmw]@{A,B})@{A,B}))
@{A,B} -> {{A,B}} ((array{{A, B}}((nat[gmw]@{A,B})@{A,B}))
@{A,B} -> {{A, B}} (nat[gmw]@{A,B})@{A,B}) @{A, B})@{A, B}
def hammingDist bs0 bs1 = par {A,B}
  let len = size bs0 in
 let hammingDistRec = (fun [hammingDistRec] i distance →
    if i == len then
      distance
    else
     let inc = mux if bs0.i == bs1.i then 0n else 1n in
    hammingDistRec (i + 1n) (distance + inc)): (nat@{A,B}
    -> {{A,B}} (nat[gmw]@{A,B}@{A,B} -> {{A, B}} nat[gmw]
      @{A,B}@{A,B})@{A,B} )@{A,B}
  in hammingDistRec 0n 0n

def main : (unit@all -> {all} nat@{A})@all
def main () = par {A,B}
  let inputA = par {A} read (array {A} nat)
  from "hamming-1k.txt" in
  let inputB = par {B} read (array {B} nat)
  from "hamming-1k.txt" in

  let shareA = share [gmw, array {A} nat : {A} → {A,B}]
  inputA in
  let shareB = share [gmw, array {B} nat : {B} → {A,B}]
  inputB in

 reveal [gmw, nat : {A,B} → {A}] (hammingDist shareA shareB)
```

Figure 4: Symphony Hamming Distance Benchmark Program

[3] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1220–1237. https://doi.org/10.1109/SP.2019.00028

[4] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. *2015 IEEE Symposium on Security and Privacy* (2015), 359–376.

[5] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[6] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *2014 IEEE Symposium on Security and Privacy*. 655–670. https://doi.org/10.1109/SP.2014.48

[7] Ian Sweet, David Darais, David Heath, Ryan Estes, William Harris, and Michael Hicks. 2021. Symphony: A Concise Language Model for MPC. In *Informal Proceedings of the Workshop on Foundations on Computer Secuirty (FCS)*.

[8] The Coq Development Team. 2022. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.5846982

[9] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. Cryptology ePrint Archive, Report 2015/1153. https://ia.cr/2015/1153.