mpC, A Programming Language for Cryptographers

mpC Team

1 Introduction

In this document we describe mpC, a first of its kind, platform-independent high-level programming language for multi-party computation protocols that supports advanced features like protocol transformations ("MPC in the head"). We believe that such a programming language has a crucial role to play in the development of the full potential of MPC in practice and in academic research.

As MPC gets more widely accepted and used, customized MPC protocols will need to be developed and updated frequently, to match specific trust assumptions, communication constraints and other idiosyncrasies of the environment in which it will be deployed. A high-level programming language allows cryptographers to unambiguously specify new protocols, without having to handle lower-level implementation details (which can be handled by the compiler of the language). On the other hand, optimizations carried out at the compiler back-end will benefit all such protocols. A high-level language also allows the development of a common protocol library resource, facilitates empirical comparison between protocols developed by different teams (executed on the same back-end, to avoid confusing implementation improvements with protocol improvements), and easy inter-operability across different deployment platforms.

A proof-of-concept compiler that supports most of the features required, and generates C++ code (which can in turn be compiled using an optimizing C++ compiler), has been developed by us. The code will be released publicly once all the core features are fully supported and adequately tested, and a basic library has been developed. The language itself, as well as protocol libraries, are designed to be easily extensible via user-defined types. Alternate compilers using other implementation techniques can also be developed in the future. Many of the low level details (message formats, party addresses, session IDs, interface with user-defined types, etc.) are not part of the language itself, but are standardized as part of our reference implementation.

mpC is a "protocol specification language" which can be used to describe/define multi-party protocols at a level of detail similar to that used in cryptographic literature. mpC separates the implementation task of security protocols into the following steps:

1. Developing and refining optimizing compilers for the language;
2. Writing protocols in the language. While protocols for basic standard functionalities can be provided as part of a library, for higher-level applications protocol generation will split again into:
   (a) Specifying a *functionality*: A functionality is simply a protocol involving a trusted party, and is easily programmed in mpC syntax.
   (b) Transforming the functionality into a protocol that avoids the trusted party. There are two kinds of transformations.
   - *Non-Blackbox Transformations:* Using "protocol schemes" (e.g., GMW, BGW etc.), which act on the code of the functionality.

- *Blackbox Transformations:* Using techniques like player virtualization and composition which use other protocols in a black-box manner.

As described below, mpC provides rich support for all these steps, targeting not only the ease of end-users (who write the functionalities), but also of the cryptographers (who program the schemes and the transformations, and may contribute to the protocol library).

2 Features of mpC

First, we present a code snippet to illustrate the fairly intuitive syntax of the language. This code illustrates some basic features of the language, like user-defined data types (e.g., `field` is a user-defined type and a subtype of it is accepted as an argument by the first two protocols, at lines 4 and 21), code blocks in individual parties using `in` (e.g., lines 7-14) and common code (e.g., lines 16-17), convenient syntax for looping using the declaration-cum-iteration operator `#` (e.g., lines 10-13) and communicating using the send-cum-assignment operator `=>` (e.g., line 17 or 44), and a compact syntax for invoking a sub-protocol using a `run` statement, with roles assigned using the `@` statements (e.g., lines 26-29). It also shows the default port `environ` used by parties to communicate with their environment and ports used to communicate with peer parties in the protocol (these ports are simply specified by the peer party's name). Additional *named* ports connecting a party to its environment can also be declared. These ports can be explicitly addressed, if necessary, though the `=>` operator can often be used to avoid the need for doing so.

While `run` is adequate for a simple secure function evaluation (SFE) protocol invocation, we also provide a more flexible mechanism to spawn off a session of a subprotocol, which can be accessed later on (e.g., for a reactive functionality). This uses a built-in data type `session`. An illustrative example of the use of a reactive subprotocol `commitment` is given in the protocol `cointoss` (see lines 35, 38, 41, 46, 50).

Protocol 1: mpC code example

```
1  uses type field {
2      field sample();
3  }
4  protocol sharing (int n, field F) : party Sndr, Rcvr[n] {
5      in each Rcvr[#i]
6          F share;
7      in Sndr {
8          F m, shares[n], sum = 0;
9          environ => m;
10         for each shares[#i] except i==0 {
11             shares[i] = F::sample();
12             sum = sum + shares[i];
13         }
14         shares[0] = m - sum;
15     }
16     for each Rcvr[#i]
17         Sndr.shares[i] => Rcvr[i].share;
18     in each Rcvr[#i]
19         share => environ;
20 }
21 protocol pointless(int m, field F) : party Clnt, Srvr[m] {
22     in Clnt
23         environ => F x;
24     in each Srvr[#i]
25         F share;
26     run sharing(m, F) {
27         @Clnt:  x => Sndr =/;
28         @Srvr[#j]:  =/ Rcvr[j] => share;
29     }
30     in each Srvr[#i]
31         share => environ;
32 }
33 protocol cointoss() :  party P, Q {
34     uses protocol commitment():  party Sndr, Rcvr;
35     open commitment session com() { Sndr@P; Rcvr@Q; }
36     in P {
37         bool b, a = bool::sample();
38         a => com.Sndr; // make committment
39     }
40     in Q {
41         com.Rcvr => nil x; // wait to receive commitment
42         bool b = bool::sample();
43     }
44     Q.b => P.b;
45     in P {
46         "reveal" => com.Sndr; // request opening
47         a+b => environ;
48     }
49     in Q {
50         com.Rcvr => bool a; // receive opening
51         a+b => environ;
52     }
53 }
```

Protocol Schemes. mpC provides a powerful mechanism to specify transformation of functionalities into protocols. Such a specification is called a `scheme`. "General MPC protocols" (BGW, GMW, Yao's Garbled Circuit based 2PC etc.) are all such schemes. Note that a scheme specifies a *non-blackbox* transformation to be applied to the given functionality. We develop a versatile syntax for scheme, using *template matching*.

The snippet to the right is from the GMW scheme. Given a protocol which has a trusted party (identified as `T` in the scheme), the scheme defines a protocol in which that party is replaced with an array of parties (`P[n]`). Whenever a field variable is declared in `T`, the scheme replaces it by a declaration of a variable by the same name in the parties `P[i]`. When `T` has a multiplication, it is replaced by a call to a protocol **`mult`** executed by the parties `P[i]`. The protocol **`mult`** (not shown) accepts additive shares of a pair of **`field`** elements from `n` parties and returns to each of them a share of the product of the two elements.

Protocol 2: Snippet of a scheme

```
1  scheme gmw<T>(int n, type field F) : party P[n] {
2      uses protocol mult(int n, type field F) : party A[n];
           ⋮
3      template (type field F) {in T F a;} {
4          in each P[]
5              F a;
6      }
7      template (F T.a, F T.b, F T.c) {in T a = b*c;} {
8          run mult(n,F) {
9              @P[#i]:  {b,c} => A[i] => a;
10         }
11     }
           ⋮
12 }
```

Another useful non-blackbox code transformation supported is "flattening." This allows one to (recursively) replace **`run`** statements with inline code of the protocol being invoked (after renaming variables appropriately), until all the remaining **`run`** statements invoke protocols whose code is not available to the compiler. Flattening can be applied to schemes as well as protocols.

Support for Blackbox Transformations. `mpC` also offers extensive features to support blackbox transformations. In principle, the main requirement of blackbox transformations is the ability to access the *next-message function* of a party in a protocol. This can be achieved if the party of interest can be executed in isolation as a protocol session, with inputs and randomness of it controlled (based on a transcript). In `mpC`, a versatile set of features provide a convenient way to implement this. Firstly, a **`partial open`** statement can be used to create a session of a protocol in which only a subset of the parties are actually run. The entire session could be private to a party, or could be shared between some parties. Secondly, a party that hosts one or more parties in a session can exercise significant control over them: it can set or read their randomness, and access or manipulate their incoming and outgoing messages. For the latter, a convenient feature is the ability to **`connect`** the ports of those parties with each other, or with ports of the hosting party. Further, reactions can be triggered when the hosted parties send or try to read from their ports (using a **`react`** block to specify the reactions). This feature is important when the hosting party is to be programmed oblivious of the protocols of the hosted parties (such as feeding input whenever the hosted party asks for it).

In Protocol 3 we show a *complete specification of the IPS transformation* that fits in about 60 lines of code (with only the protocol for a fixed watchlist functionality not shown). This transformation takes an outer protocol and an inner protocol that are assumed to have already been created.[1] Note that the protocol **`IPS`** takes these two protocols as input arguments, let its client parties host the client parties in a *partial* session of the protocol **`outer`**, with the servers in that session replaced by sessions of the protocol **`wrapinner`** (to which the protocol **`inner`** is passed as an argument). This protocol demonstrates the usefulness of

[1] For simplicity, the inner protocol is assumed to be in the plain model, and not invoke any subprotocols. See the flattening feature above to get this.

partial open combined with the ability to wire ports together using **connect**. The protocol **wrapinner** demonstrates the usefulness of the **react** statement to associate actions with port activity.

Protocol 3: IPS Transformation

```
protocol wrapinner (int nC, int nS, int index, field F, (protocol inner(int n, int index, field F) : party P[n]))
       :  party R[nC] each with {outport Wrandout, outport Wout, inport Wrandin[nC], inport Win[nC]} {
   in each R[#i] { // implement P[i] of inner, possibly verifying the others.
      partial open inner session Z(nC,index,F) { P[i]@self; }
      connect Z.P[i] to environ;
      forward Z.P[i]|random to Wrandout; // Report randomness on watchlist channel
      connect environ to { Wout, Z.P[i] }; // "Tee" inputs to watchlist channel
      start Z;   // partial open requires explicit start
      Win[i] => bool status; // status = is watchlist channel open to R[i]
      inner session Q; // to run a shadow session if watchlist is open
      if (status) {
         partial open Q(nC,index,F) { for each Q.P[#j] except j==i P[j]@self; }
         for each Q.P[#j] except j==i {
            forward Wrandin[j] to Q.P[j]|random; // set randomness (allowed before start)
            connect Win[j] to Q.P[j]|environ; // pass on inputs
         }
         start Q;
      }
      react { // Relay messages between Z.P[i] and peers R[j]
         before each p of Z.P[i]|P[#j] except j==i {
            R[j] => nil y;
            if (status) { // Verify message from R[j] matches prediction by Q.P[j]
               Q.P[j]|P[i] => nil x;
               if (x!=y) abort;
            }
            y => p;
         }
         after each p of Z.P[i]|P[#j] except j==i {
            p => nil y;
            y => R[j];
            if (status)
               y => Q.P[j]|P[i];
         }
      }
   }
}
protocol IPS (int nClnts, int nSrvrs, field F,
              (protocol outer(int nC, int nS, field F) : party Clnt[nC], Srvr[nS]), (protocol inner(int n, int index, field F) : party P[n])
              ) :  party C[nClnts] {
   uses protocol watchlist (int nC, int nS, field F)
              :  party P[nC] each with {inport WLrandreport[nS], inport WLreport[nS], outport WLrandread[nS][nC], outport WLread[nS][nC]};
   wrapinner session In[nSrvrs]; // One wrapinner session per outer server
   for each In[#s]
      open In[s](nClnts,nSrvrs,s,F,inner) { R[]@C[]; }
   open watchlist session W(nClnts,nSrvrs,F) { P[]@C[]; }
   // connect wrapinner ports to watchlist
   for each In[#s]
      in each C[#i] {
         connect In[s].R[i]|Wout to W.P[i]|WLreport[s];
         connect In[s].R[i]|Wrandout to W.P[i]|WLrandreport[s];
         for each win of In[s].R[i]|Win[#j]
            connect W.P[i]|WLread[s][j] to win;
         for each wirandn of In[s].R[i]|Wrandin[#j]
            connect W.P[i]|WLrandread[s][j] to wrandin;
      }
   // C[i] will host Clnt[i] of an outer session (in which servers are not hosted)
   partial open outer session Out(nClnts,nSrvrs,F) { Clnt[]@C[]; } // no servers
   in each C[#i] {
      for each o of Out.Clnt[i]|Srvr[#s]
         connect o and In[s].R[i];    // connect with In[s].R[i] instead of Srvr[s]
      connect environ and Out.Clnt[i];
   }
}
```

User-Defined Types. In the protocols above the type **field** is *not* natively supported by the language (in particular, it is not a keyword). Instead, it is a user-defined type. As our implementation uses C++, all user-defined types are written in C++, inheriting from a base-class from which standard types also inherit. Non-standard functions provided by a type (e.g., `sample` in **field**) need to be declared in the protocol uses clause, to allow the `mpC` compiler to not flag their use as errors. We allow types to have sub-types (e.g., GF(p) can be implemented as a sub-type of field, specifying p at the time of instantiating the sub-type). Note that types (or sub-types) themselves can be passed as arguments to protocols.

Importantly, user-defined types can be used as a mechanism to provide access to cryptographic prim-

itives (like pseudorandom functions or public-key encryption schemes), to resources like pre-computed correlations stored in the file-system, or even generic access to a file-system (e.g., by providing a file type).

Accessing Persistent Sessions. mpC also supports "joint-state" protocols. The importance of joint-state protocols is that "setups" are often modeled as trusted parties that can be accessed by multiple sessions. We allow a single persistent session to be "joined" from various other sessions, with each party of the persistent session temporarily hosted by a party in the joining session (provided the two are in the same "location"). To facilitate this, we allow `session` variables (of persistent sessions) to be passed as an argument to protocols.

Potential Features for Future Versions. The following are a few features that are not incorporated in the current implementation, but will be of value in the language.

- Current support for user-defined data-types is somewhat rudimentary. Full-fledged support for classes could be included in a future version of mpC (or mpC ++). Further, more data-types can be made available as part of a standard library.
- Currently, the primary execution model is an asynchronous model (with the ability for parties to configure read timeouts). A future version can natively support a synchronous execution model, relying on the availability of (approximately) synchronized clocks at all the parties.
- A future version can support automatic formal verification of various safety properties of protocols written in a subset of the mpC language (after incorporating extensions that can be used to summarize the input/output behavior of protocols).
- A future version may incorporate annotations for security models/assumptions into our protocol syntax, to facilitate composition, by finding appropriate protocols to replace a functionality, subject to user-specified requirements (e.g., an implementation of the OT functionality that is UC-secure in the random oracle model).

2.1 Implementation Notes

We have implemented a compiler for mpC, which translates mpC code into C++ code. Our compiler itself is written in Python, using ANTLR for generating the parser. The compiler carries out significant semantic checks to help catch coding bugs and errors, before producing C++ code. The C++ code produced makes extensive use of standard Boost libraries and pthreads (each party is a thread, with appropriate code loaded dynamically when the party is created). Thread-safe data structures are used to maintain the state of all the parties, ports, connections, communication buffers, trigger buffers (to support `react`), etc., at each location. A thread-pool can be optionally enabled, in which case the system limits concurrency when short on threads, and instead prioritizes completing the execution without entering into deadlocks caused by non-availability of threads.

Interfacing with External Programs. Current implementation provides basic support for interfacing with external programs and protocols. There are two ways one could use an external protocol for a functionality like, say, OT. Firstly, the external protocol can be used to pre-compute values that are stored locally at each location; then, a protocol with a persistent session can be used to access the pre-computed values thread-safely. Alternately, a C++ wrapper can be written to interface with the external protocol, so that it can be accessed just like accessing a protocol created by the mpC compiler.

Currently, external (distributed) programs can invoke mpC protocols by creating appropriate configuration files and launching the mpC engine at each location. In future, an API may be provided to simplify this.

Standardization of Formats. Data storage and communication formats need to be standardized to enable different tools to interoperate (e.g., an mpC program that uses correlated keys generated using a different tool). Future versions of mpC may include proposals for such standards, when appropriate.

3 Contributors

mpC has received inputs from several contributors. The initial discussions on mpC were carried out at Microsoft Research Bangalore, in Summer 2017. This discussion involved Manoj Prabhakaran and Sanat Anand from IIT Bombay, and Divya Gupta, Rahul Sharma, Aseem Rastogi and Nishant Chandran at MSR. Subsequent development of the language was carried out at IIT Bombay mainly by Rajeev Raghunath (2018-) and Manoj Prabhakaran, with contributions from Kartik Singhal (2019-20), Onkar Deshpande (2020-21), Shivam Goel (2021-22), Hemant Chodipilli (2021-22), Mayank Kakad (2022-23) and Neeraj Jadhav (2023-24).