Private Computation Framework 2.0 White Paper

Background

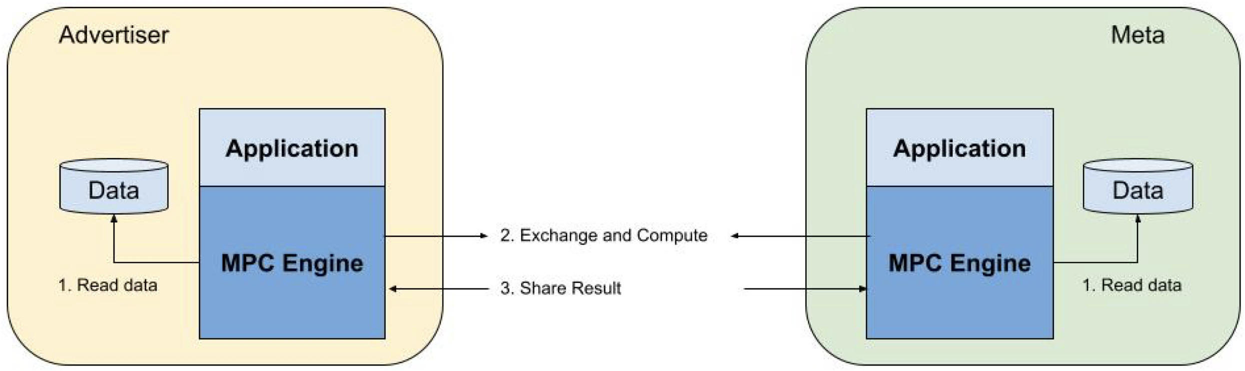
In recent years, there is an ongoing shift in the advertisement industry to enhance user privacy protection. Meta is working to build the next generation platform that enhances privacy and empowers advertisers to maximize their values. We develop a Private Computation Framework (PCF) that implements Multi-Party Computation (MPC) protocols and execution engines, and provides programming interfaces.

What is MPC

PCF is built around MPC, which is a term that describes a secure method that allows parties to jointly compute a function over their inputs while keeping those inputs private. With MPC, people's personal data remains private while still allowing platforms and publishers to collaborate with advertisers to make ads more personal and effective.

As the name of MPC suggests, the computation happens between multiple parties. Therefore, the end-to-end computation happens among multiple computation nodes. Each node is owned by a different party - e.g. Meta and advertisers - and is deployed in their separated and isolated accounts.

The following diagram illustrates how multiple parties jointly compute the data. Note that both parties use the same application (i.e. business logic) but use different data from their own end. The data on each side is private to each party and cannot be revealed to the other party. During the joint computation, each party will share encrypted data that the other party cannot decrypt. In the end, both parties will get a shared result that each party can understand (i.e. in plaintext).



History of PCF

In 2021, we developed PCF 1.0, which depended on the garbled-circuit-based semi-honest two-party-computation protocol by using the open source EMP-Toolkit library. PCF 1.0 exposed protocol-specific implementation to the application layer, and made the application hardcode with EMP interfaces and functions.

There are several limitations with this architecture and makes it difficult to:

- Adding additional MPC protocols without affecting the applications built on top of PCF.
- Adding new features into the existing PCF libraries.
- Testing and debugging the application without invoking the underlying MPC protocol.

One of our main motivations for PCF 2.0 is to decouple the iteration of applications from iterations of the framework. Another motivation is to allow the addition of new MPC protocols into the framework without revamping the applications.

What's New in PCF 2.0

Secret-Sharing-Based MPC Protocol
PCF 2.0 implements the XOR-secret-sharing based protocol (the classical GMW protocol combined with beaver tuples/multiuplicative tuples). This protocol has many advantages over the garbled-circuit based protocol that we implemented in PCF 1.0. For example, the network traffic between the computation parties is reduced by close to 100x.

Easier Application Development
PCF 2.0 is extensible by providing rich programming interfaces to application developers and abstracting away the backend protocol implementations. It allows the same application to run with different underlying protocols.

PCF 2.0 implements different execution engines that are used for different scenarios. For example, other than the production execution engine that executes the actual protocol, PCF 2.0 also implements an execution engine that executes the application logic in plaintext. It allows developers to quickly iterate and validate their business logic or debug their issues by foregoing lengthy, computationally intensive processing and reducing expensive network traffic

Reading Guide

The rest of this white paper is divided into several chapters. The Architecture chapter provides a high-level overview of the PCF architecture and a brief introduction to each PCF component. The following chapters describe each PCF component in more in-depth detail.

To ease our writing, we will use **PCF** through the rest of this document to refer to **PCF 2.0**, unless called out explicitly.

Architecture

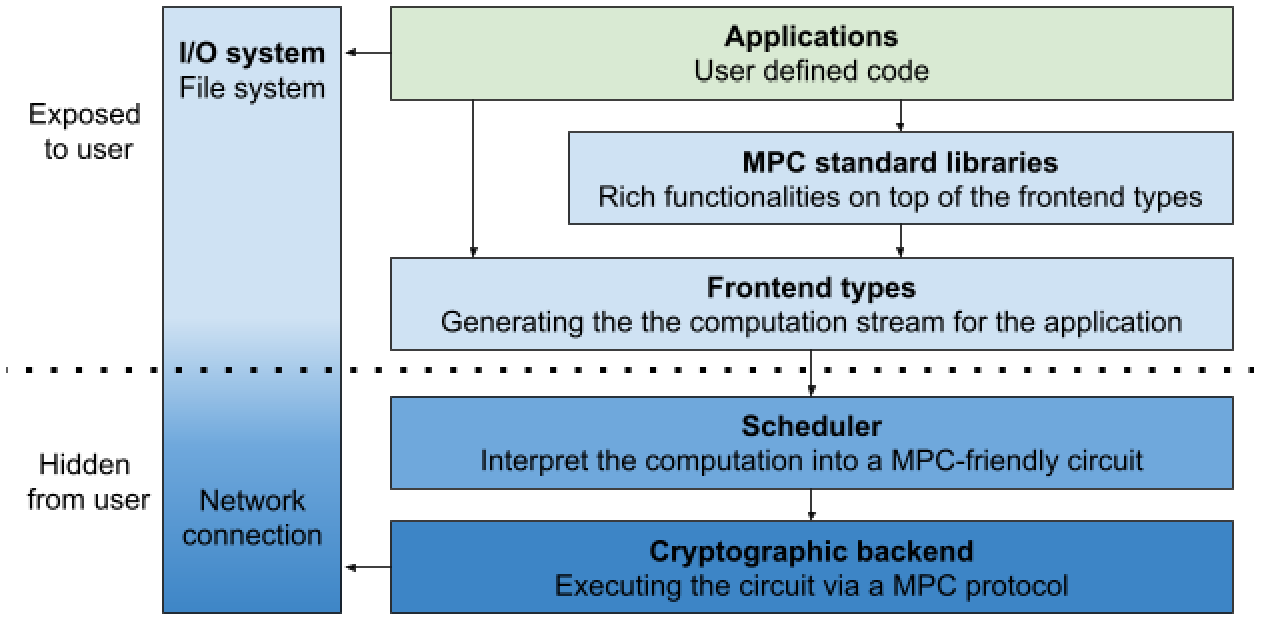
PCF builds a scalable, secure, and distributed private computation platform to run computations on a production level. PCF supports running the computation on a public cloud.

PCF consists of five major components:

- **Frontend Types** provide the basic types and operators to the application developers as their programming interface.
- **MPC Standard Libraries** provide useful features like ORAM, shuffler, sorter, similar to standard libraries of common programming languages.
- **Scheduler** translates the application written in high-level language into its equivalence, represented in a MPC-friendly format.
- **Cryptographic Backend** implements the desired MPC protocols and executes the business logic under the protection of those protocols.
- **I/O system** connects participating parties together and provides the file system service.

In an analogy, PCF is like a computer system. The frontend types and the MPC standard libraries are the user-facing components, which work like the input devices, conveying the user’s intent to the rest of the system. The cryptographic backend plays the role of the CPU, which is responsible for executing the user's commands eventually. The I/O system handles the external shortages. The scheduler is like the motherboard, hosting the cryptographic backend and translating the user's commands into instructions for the cryptographic backend/CPU to execute. The scheduler also hosts all the utility functionalities like bookkeeping and garbage collection.

The diagram below demonstrates the interactions among the user-defined applications and the major components of PCF.


Exposed to user
Hidden from user
I/O system
File system
Network connection
Applications
User defined code
MPC standard libraries
Rich functionalities on top of the frontend types
Frontend types
Generating the the computation stream for the application
Scheduler
Interpret the computation into a MPC-friendly circuit
Cryptographic backend
Executing the circuit via a MPC protocol

1. The application developer implements the business logic by using C++. They program against the **Frontend Types and MPC Standard Libraries**, which provide basic types, operators, and common functionalities. These types, operators, and functions will be converted to internal types and consumed by the **Scheduler**.
2. The application can access storage via the abstracted functions provided by the **I/O System**.
3. **Scheduler** takes the converted Frontend types and operators and translates them into MPC-friendly instructions that can be consumed by the **Backend**. The scheduler is also responsible for book-keeping all types of resources and any garbage collection tasks.
4. **Backend** implements all cryptographic MPC protocols. It receives the MPC-friendly instructions from the scheduler and executes them collectively among multiple computation parties. The communication happens via the network implemented by the **I/O System**.

PCF is currently implemented in C++. The following sections describe each PCF component from a high-level. More in-depth descriptions will be provided in later chapters.

Frontend Types

PCF implements **Bit**, **Int**, and **BitString** types. They are the equivalent of **bool**, **int**, and **string** in native C++. Application developers implement their business logic by programming against these frontend types with other native C++ functions. The frontend types are compatible with most native C++ operators and standard library functions, but certain limitations apply.

The frontend types connect the user-defined application and the scheduler by acting as the coordinator of various behind-the-scenes components. Operations on those frontend types are translated into a sequence of calls to low-level functions that will be consumed by the schedulers. This translation may happen recursively. For example, an operation on the PCF **Int** will be translated into a set of operations on the PCF **Bit** that composes the integer object.

MPC Standard Libraries

Due to the frontend type's limited compatibility with C++ standard library functions, realizing certain high-level functionalities (e.g. group-by) becomes a daunting task, and often results in unacceptable performance. A set of MPC standard libraries are developed to fill in the blanks. Currently we have two functionalities: ***write-only oblivious RAM*** and ***oblivious shuffle***. We will explore this in more detail later in the paper.

I/O System

PCF has two main I/O components:

- File I/O
- Network I/O

The PCF I/O system provides layered APIs to access different types of data sources or sinks, where the application developers program against well-abstracted I/O functions without worrying about the underlying file and network system and implementation.

Scheduler

The scheduler bridges the frontend types and the cryptographic backends and facilitates multi-threading. It translates the gate stream generated by the frontend types into a MPC-friendly circuit via topological sorting. It also bookkeeps wires and gates in this circuit and passes the gates to the cryptographic backend for execution following the topologically-sorted order.

Multiple scheduler instances, indexed by a scheduler **id**, can exist at the same time. This allows multi-threading in PCF. Scheduler instances exist in the global scope. Their lifespans last till the end of the program after all private computation finishes. Frontend types pass in a template parameter **schedulerId** to decide which scheduler instance should accept its gate stream. Each scheduler instance has an exclusive ownership of a cryptographic backend instance. This exclusivity guarantees the monopoly mapping from frontend to backend. This design allows multiple MPC parties to exist concurrently. This feature can be used for in-process MPC tests in development and multi-threading for better performance in production.

Cryptographic Backend

The cryptographic backend implements various cryptographic protocols. It consumes the computation's instructions, along with the necessary input data, from the scheduler. After executing the computation per scheduler's instruction, the results are returned to the scheduler for further processing.

Application developers are not expected to interact with the backend other than creating the instances at the beginning of their program (i.e. selecting the computation protocol).

Frontend Types

The PCF Frontend encapsulates MPC types, operators, and functions in templated C++ implementations. In analogy, PCF Frontend provides a "programming language syntax" that allows PCF application developers to easily implement their business logics without being exposed to the underlying cryptographic protocols or infrastructure. This design provides good abstraction between different software layers and hence allows great extensibility and scalability of the underlying software layers (eg. PCF Scheduler and PCF Backend).

Programming Interface

The frontend types are the base classes that support all the objects offered by PCF. The following object types are provided in PCF Frontend: **Bit, Int,** and **BitString**, corresponding to the vanilla C++ types of **bool, int,** and **string**. These types are templated with different template parameters controlling the behavior of each type.

These PCF frontend types are implemented as C++ template classes:

```
template <bool isSecret, int schedulerId, boo usingBatch>
class Bit {...}

template <bool isSigned, int8_t width, bool isSecret, int schedulerId, bool
usingBatch>
class Int {...}

template <bool isSecret, int schedulerId, bool usingBatch>
class BitString {...}
```

The common template parameters are listed below:

Common Parameter	**Usage**
`bool isSecret`	Indicate whether this variable represents a secret value. ● Secret value is encrypted and cannot be extracted by any party that doesn't own the permission to the original value. ● Public value is readable by all parties.
`int schedulerId`	Indicate which scheduler this variable should connect with. More about Scheduler in a later chapter.

`bool usingBatch`	Indicate whether this is a batched type. More about batching in a later section.

In addition to the common template parameters, some types have their own template parameters:

Type	Parameter	Usage
Int	`bool isSigned`	Indicate whether this is a signed integer or an unsigned integer.
	`int8_t width`	Indicates the width of bits that compose this integer.

The frontend types are compatible with most C++ features, including containers and some C++ standard library functions. However, some native C++ functions are incompatible. For example:

- C++ containers like std::vector and std::array can work with PCF types perfectly.
- C++ containers like std::map can use PCF types as the value but not the index.
- All the comparison operators (e.g. `==, <`) on the PCF frontend types are overloaded. The output of the comparison operations on the PCF frontend types is PCF **Bit**. In contrast, native C++ comparison operators return **bool**. This means that application developers cannot directly use the comparison output of PCF frontend types with the native C++ conditional statements like **if, while, switch**, etc. For example:

```
// Native C++ type
int a = 10;
int b = 8;
if (a > b) {
  // Valid comparison
}

// Omitting the template parameters for this example
frontend::Int x(10);
frontend::Int y(8);
if (x > y) {
  // Invalid comparison. This won't compile.
  // x > y results in a frontend::Bit object, instead of native C++ bool
}
```

- C++ standard library functions like std::sort() and std::find_if() do not work with PCF frontend types because their internal comparison implementation expects vanilla C++ **bool**, not PCF **Bit**.
- The PCF **Int** can't be used in places where native C++ **int** type is expected, for example serving as the index to access any C++ container.

To compensate for the loss of functionalities, we implement a list of utility functions and MPC standard libraries. More details of the libraries are in later chapters.

Internal Working Mechanism

Each frontend type implements operators that are mostly isomorphic to their native C++ counterparts. For example, an XOR operation between two Bit type inputs results in a Bit type output, similar to an XOR operation between two C++ bool type inputs resulting in a bool type output.

Each frontend type also implements methods that facilitate the translation of data between application space and PCF space. For example, the instantiation of frontend::Int(8) encapsulates the native C++ integer of value 8 into an PCF Int object that is implemented as a **gate stream of a circuit**. The gate stream can then be consumed and processed by other PCF components like scheduler and cryptographic backend. The Int object can be encrypted so that the value is invisible to the counter-computation-party. Once the computation finishes, the application can call other PCF functions to decrypt the computation result out of the Int object and use the value as native C++ integer.

Common Methods

All frontend types implement constructors that accept their counterpart native C++ types as input. The template parameter **isSecret** of each type indicates whether the value should be encrypted by the backend or not. If isSecret is false, the value is public and hence is readable by every computation party. Otherwise, the value is private and is only accessible by the party who instantiates the frontend type object. This party must supply a partyId to the constructor to indicate its ownership, and should keep the partyId a secret.

Secret values can be converted to storable shares for storage via the extract functions (e.g. extractBit() for Bit type). A stored secret can be recovered by having each party provide their own share of the stored secret.

Secret values can be made public by opening it to a party, which later can be accessed via a call to getValue`()`. Only the secret-owner party can get the actual value while other parties get a dummy value. This public value shouldn’t be used for further computation unless the application explicitly synchronizes it across all participants (e.g. by opening the same secret value to everyone).

Bit

We support the following operators on `Bits`: `!, &, ^, |`. These operators work by generating gates for the scheduler. Specifically, the `!` operator is converted into NOT gates, the `&` operator is converted into AND gates, and the `^` operator is converted into XOR gates. The `|` operator is implemented using the `^` and `&` operators.

```
// For this example, we omit some template parameters for ease of writing.
frontend::Bit foo(true);
frontend::Bit bar(false);

frontend::Bit notFoo = !foo; // notFoo.getValue() returns false
frontend::Bit andFooBar = foo & bar; // andFoBar.getValue() returns false
frontend::Bit xorFooBar = foo ^ bar; // xorFooBar.getValue() returns true
frontend::Bit orFooBar = foo | bar; // orFooBar.getValue() return true
```

Integer

Just like how native C++ int can be represented as arrays of bits, PCF Int is implemented as an array of PCF Bit. Operations on PCF Int type are converted to PCF Bit operations. PCF supports the following operators on PCF Int: `+, -, <, <=, >, >=, ==,`.

```
// For this example, we omit some template parameters for ease of writing.
frontend::Int foo(8);
frontend::Int bar(13);

frontend::Int sum = foo + bar; // sum.getValue() returns 21
frontend::Int sub = foo - bar; // sub.getValue() returns -5

// Note that comparison operations returns a frontend::Bit object instead
of a native C++ bool
frontend::Bit isLess = foo < bar; // isLess.getValue() returns true
frontend::Bit isLessOrEqual = foo <= bar; // isLessOrEqual.getValue()
returns true
frontend::Bit isMore = foo > bar; // isMore.getValue() returns false
frontend::Bit isMoreOrEqual = foo >= bar; // isMoreOrEqual.getValue()
returns false
frontend::Bit isEqual = foo == bar; // isEqual.getValue() returns false
```

As stated in the example above, because the comparison operation of PCF Int types results in PCF Bit type, the native C++ conditional statements do not work directly with the overloaded comparison operations. To facilitate application development, PCF implements a `mux` (multiplexer) function.

```
// For this example, we omit the template parameters for ease of writing.
```

```
frontend::Int foo(8);
frontend::Int bar(13);
frontend::Bit condition = foo > bar;

// The mux function evaluates the condition foo > bar, which is
encapsulated in a Bit object.
// If the evaluation is false, it returns foo (the caller of mux).
// If the evaluation is mux, it returns bar (the input of mux).
// In this example, the condition is Bit(false). Hence, result = bar.
frontend::Int result = foo.mux(condition, bar);
```

BitString

The frontend `BitString` type represents a vector of bits, with some basic bitwise operations. Internally, PCF implements `BitString` as a vector of `Bits`. PCF supports the following operators on `BitStrings`: `!, &, ^`.

Examples:

- !BitString({true, false, true}) -> BitString({false, true, false})
- BitString({true, false, true}) & BitString({false, true, true}) -> BitString({false, false, true})
- BitString({true, false, true}) ^ BitString({false, true, true}) -> BitString({true, true, false})

Similar to PCF Int, PCF BitString also implements a `mux` (multiplexer) function to facilitate development of conditional logic.

```
// For this example, we omit the template parameters for ease of writing.
frontend::Bit condition(13 > 8);
frontend::BitString foo({true, false, true});
frontend::BitString bar({false, false, true});

// The mux function evaluates the condition 13 > 8, which is encapsulated
in a Bit object.
// If the condition is false, it returns foo (the caller of mux).
// If the condition is true, it returns bar (the input of mux).
// In this example, the condition is Bit(true). Hence result = foo.
frontend::BitString result = foo.mux(condition, bar);
```

Batching and Re-batching

As described above, frontend types generate a gate stream for the scheduler to process. The same piece of code may execute with different data, which results in repeatedly generating and sorting the same gate stream. PCF allows application developers to explicitly mark down code that will execute repeatedly with large amounts of data to avoid duplicate gate stream generation and hence improve performance. This is achieved via the batching mechanism.

All PCF frontend types have a **usingBatch** template parameter. When true, a frontend type can be viewed as a vector of the original type. For example, a `Bit` type with `usingBatch` set to true works just like a vector of `Bit` with that set to false. When an operation is performed on a batching type, it will be propagated to each element in the (conceptual) vector. Binary operations can only be performed between two batching vectors of the same size and it will be interpreted as element-wise binary operations, resulting in a (conceptual) vector as outcome, stored in a new value of a batching type.

```
// For this example, we omit some other template parameters for ease of writing.
// Note that with different usingBatch values (true or false), the input to the constructor.
// and the output of getValue() are different types: vector<int> and int, respectively.
// The input/output types are templated, depending on the value of usingBatch.

// When usingBatch is true, the frontend Int type can accept a vector of integers as its constructor input.
frontend::Int<usingBatch=true> foo(std::vector<int>({8, 10, 13}));
frontend::Int<usingBatch=true> bar({5, -2, 7});
frontend::Int<usingBatch=true> sum1 = foo + bar;
std::vector<int> result1 = sum1.getValue();
// result1 is a vector that contains {13, 8, 20}

// When usingBatch is false, the frontend Int type can accept a native int as its constructor input.
frontend::Int<usingBatch=false> tea(8);
frontend::Int<usingBatch=false> pot(9);
frontend::Int<usingBatch=false> sum2 = tea + pot;
int result2 = sum2.getValue();
// result2 is an int of 17
```

This batching feature comes with the option of rebatching. Rebatching APIs allows splitting a big batch into multiple smaller ones or vice versa.

Scheduler and Scheduling Algorithms

The scheduler sorts the gate stream from frontend types into a MPC-friendly order while keeping the circuit it represents unchanged. The sorted gate stream is passed to the cryptographic backend for execution. In addition, the scheduler bookkeeps the intermediate value during the computation, aka the wires in the circuits. The bookkeeping mechanism is implemented via memory arenas for best performance.

Bookkeeping Algorithm

Variables of frontend types have their own life spans and are assigned with multiple values. However, this is not the case for circuit wires. Each circuit wire can only get exactly one value from the gate that creates it. A circuit wire can be deallocated only if no future computation would possibly involve it. PCF adopts the strategy from a related paper used for the same purpose and implements it into a wire keeper.

A many-to-one mapping is maintained during the execution. Each variable of frontend types is mapped to a circuit wire kept inside the wire keeper. Those variables store the index of the wires they are mapped to, but not the values associated with that wire. The wire keeper object keeps running sums of how many variables are mapped to each wire. Once the last variable that maps to a particular wire goes out of scope, this wire will be up for deallocation.

Plaintext Scheduler

As a development tool, our basic scheduler implementation is a plaintext scheduler. There are two versions of it: a normal one and a networked one, for testing in different environments. This type of implementation doesn't really connect to a cryptographic backend. Instead, all computations are executed in plaintext. These schedulers can be used for development purposes and avoiding invoking the heavy-weighted cryptographic backend when it is irrelevant.

Eager Scheduler

This is a lightweight scheduler that doesn't sort the gate stream but simply passes it to the cryptographic backend. This implementation can be performant only if the underlying cryptographic backend (and the protocols it implements) doesn't have preferred gate order. An example use case would be when the cryptographic backend implements a garbled circuit based protocol that is not sensitive to circuit depth.

Lazy Scheduler

Some underlying cryptographic protocols are sensitive to circuit depth (e.g. the secret-sharing-based protocols). An eager scheduler could theoretically work, but it will likely incur a large amount of unnecessary circuit depth due to not sorting the gate stream. To address this issue, we adopt and modify the algorithm from a research paper to topologically sort the gate stream before passing it to the cryptographic backend for execution.

In the underlying backend, some types of gates are executed without communication between parties in the underlying cryptographic protocol while others rely on cross-party communication. Thus gates are classified into two categories: free gates and non-free gates. Executing secret-sharing-based MPC protocol (like the one we implemented) incurs a roundtrip for each non-free gate. To mitigate the latency from extra roundtrips, non-free gates should be executed in parallel as many times as possible. In other words, the topological sorting algorithm should concrete parallelable non-free gates to enable batching.

Our sorting strategy can be summarized as labeling each gate a dynamically calculated level, and sorting gates according to their levels. We set even levels to be levels of free gates and odd levels to be levels of non-free gates. The level of a free gate is the lowest free gate level that is at least as high as all of its input wires' levels. The level of a non-free gate is the lowest non-free gate level that is higher than any of its input wires' levels. The gates of each level are buffered in a queue, such that the relative order in each level is preserved after the topological sorting.

When a gate stream comes into the lazy scheduler, new gates are pushed to the back of queues of the corresponding level. The buffered gates are executed if certain conditions are met: the application requests plaintext results; the number of buffered gates reaches a pre-set threshold; or reaches the end of the gate stream.

Debugging with Schedulers

Various scheduler implementations help debug applications in various ways. The plaintext schedulers can be used to verify the correctness of the app itself without invoking the cryptographic backend, either piece-by-piece or end-to-end. The eager scheduler can be used to test applications with the cryptographic backend without the complicated scheduling algorithm, helping to locate errors faster.

Cryptographic Backend

The cryptographic backend is the combination of all the underlying cryptographic protocols. It provides the interface to execute types of gates following concrete private computation protocols. Currently we have a XOR-secret-sharing-based MPC protocol. An arithmetic-secret-sharing-based protocol is a future area of interest.

The Secret-Sharing-Based MPC Protocols

Currently our focus is the secret-sharing-based MPC protocol for its potential performance. Our target security model is the semi-honest model with a dishonest majority. In the future, we will investigate support for stronger models.

At a high-level, the secret-sharing-based MPC can be described as following:

1. Converting the computation to a circuit of some basic gates.
2. Secret-sharing all the input data.
3. Perform the computation on the secret-shared inputs by traversing the circuit, each gate in the circuit will take in secret-shared signals on input wires and generate secret-shared output signals.
4. Recovered secret-shared outputs.

XOR-Secret-Sharing-Based MPC

At the core of the framework sits the SecretShareEngine class which executes the computations that are passed to it by the scheduler. The operations that are supported by the SecretShareEngine are the boolean operations XOR (^), AND (&), and NOT (!). This is enough to represent any calculation that can be done by a turing machine.

The values that are operated on can be public or private. So for the XOR operation, because it is binary, it will have 3 versions (PublicXORPublic, PrivateXORPublic, PrivateXORPrivate). In the case of a public value, both parties are expected to contain the same bit in their storage. This means that a NOT on a public value or an AND of two public values can directly be computed by each party with no communication involved.The new value will be returned to the scheduler and then stored to be used in further calculations.

In the case of a private value that is unknown to either party, each party will have one share of the value. A secret share is simply a bit that can be used to recover the original value by taking all the parties shares and XOR'ing them together. For example, if there are 3 parties A, B, and C with the shares False, True, False, then they can recover the share value by taking XOR(XOR(False, True), False) = True. It's important to note that since XOR is both commutative and associative that the order they combine the shares in does not matter, the result is always the same. Furthermore, the MPC Protocols guarantee that knowledge of your secret share of a value has no correlation with the private value, i.e. the distribution of the other parties secret shares added up to a certain bit should be uniform.

With this definition in mind, the Secret Share Engine is in charge of calculating the updated secret shares following one of the boolean operations listed above.

The XOR operation is fairly simple in that it doesn't require any communication between the two parties. Let u and v be two secret shared bits that are private to each party. Each party has its own two bits (u_A, v_A) and (u_B, v_B) respectively. They can recover the values of u and v by taking the XOR of their shares (i.e. $u = u_A \oplus u_B, v = v_A \oplus v_B$). Then setting $w_A = u_A \oplus v_A$, $w_B = u_B \oplus v_B$ will give each party secret shares of the value $w = u \oplus v$, while neither party needs to know w, u or v. If one of the values is public while the other is private, then just one party will have to XOR its share with that value while the other party re-uses the same value.

Similarly as above, the NOT operation does not require any communication between the two parties. In the private value case one party needs to flip its share, thus $w_A =! u_A$, $w_B = u_B$, and $w = !u$. If the value is public then all parties will update their value locally.

For the AND operation on two secret shared values the parties will need to do something more tricky which involves some communication. The parties have secret shares of $u = u_A \oplus u_B$ and $v = v_A \oplus v_B$. To compute the secret shares w_A and w_B such that $w = u \,\&\, v$ they will need the help of a single-use multiplicative tuple that has been generated ahead of time. The idea of the multiplicative tuple is that it is a set of secret shared values a, b, c such that $a \,\&\, b = c$. Neither party knows the values of a, b, c. Instead they know the secret shares which form the equation$(a_A \oplus a_B) \,\&\, (b_A \oplus b_B) = c_A \oplus c_B$. They will use these values to mask their shares of u and v and reveal the values $u \oplus a$ and $v \oplus b$ by exchanging the bits $u_A \oplus a_A$, $v_A \oplus b_A$, $u_B \oplus a_B$, $v_B \oplus b_B$. Finally they compute the shares of w by setting

i. $w_A = c_A \oplus (u \oplus a)b_A \oplus a_A(v \oplus b) \oplus (u \oplus a)(v \oplus b)$

ii. $w_B = c_B \oplus (u \oplus a)b_B \oplus a_B(v \oplus b)$

It is simple to show that

$$\begin{aligned} w_A \oplus w_B &= (c_A \oplus c_B) \oplus (u \oplus a)(b_A \oplus b_B) \oplus (a_A \oplus a_B)(v \oplus b) \oplus (u \oplus a)(v \oplus b) \\ &= ab + (u \oplus a)b + a(v \oplus b) + (u \oplus a)(v \oplus b) = uv \end{aligned}$$

Generate Multiplicative Tuples

A key step in a secret-sharing-based MPC protocol is generating the multiplicative tuples that can be used for evaluating And/multiplication gates. In PCF, we use oblivious transfers to generate all kinds of tuples. We have a general solution for an arbitrary number of parties and an optimized special solution for the two-party scenario.

General Solution

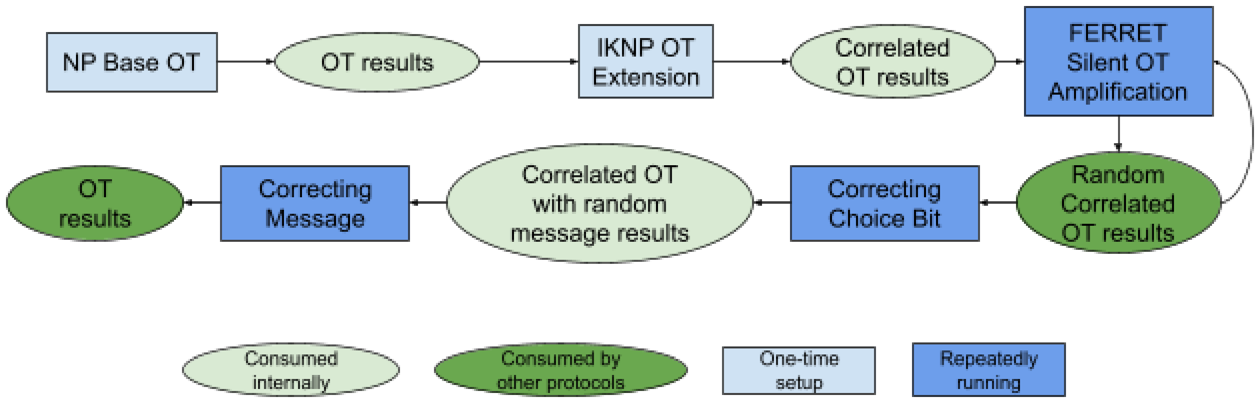
Generating a multiplicative tuple for n parties, is equivalent to to compute the secret-shares for $\left(\sum_i a_i\right) \cdot \left(\sum_i b_i\right)$ where party i holds a_i and b_i. This can be done by letting each pair of party, party i and party j, running two OTs to compute the secret-shares of $a_i \cdot b_j$ and $a_j \cdot b_i$ respectively. Note that an implication here is party i needs to use the consistent a_i and b_i when interacting with different party j. Therefore more communication-efficient random OT protocols can not be used here.

Optimization for Two-Party Case

The consistency issue in the generation scenario vanishes when there are only two parties. The two parties can run a random OT such that party 1 gets random m_0, m_1 as OT sender and party 2 gets random $c,\ m_c$ as OT receiver. Party 1 then set a_1 to $H(m_0) \oplus H(m_1)$ and his share of $a_1 b_2$ to be $H(m_0)$. Party 2 then set $b_2 = c$ and his share of $a_1 b_2$ to be $H(m_c)$. Then the two parties use the same approach to compute secret shares of $a_2 b_1$, followed by calculating the shares of $\left(a_1 \oplus a_2\right) \cdot \left(b_1 \oplus b_2\right)$ out of shares of $a_1 b_2$ and $a_2 b_1$.

Oblivious Transfers

Variants of oblivious transfers is the core protocol for tuple generation. Our roadmap of efficiently instanting different variants is shown below. We have a base OT protocol and an OT extension protocol to bootstrap the FERRET protocol to generate random correlated OT (RCOT) results. Then those RCOT results can be converted to OT results when necessary.



FERRET Protocol

We adopt the FERRET protocol to generate random correlated OT results. This protocol needs thousands of correlated OT results to bootstrap. We choose the classical NP OT protocol and the semi-honest version of IKNP OT extension protocol as the bootstrapper to generate the initially needed correlated OT results. In our implementation, we make the assumption of LPN with regular error that provides a security guarantee in the presence of a semi-honest adversary.

Optimization for Special Gadgets

Theoretically, it is sufficient to realize arbitrary computation from some basic types of gates. Optimization for certain special gadgets can actually significantly reduce the total overhead, especially when usage of such gadgets is massive. We currently have developed support for composed AND gates as the first step and research support for others in the future.

Many to One AND Gates

In some typical gadgets, e.g. multiplexers, multiple AND gates may use the same bit as one of their inputs. Since the AND operation is realized by leveraging multiplicative tuples, this observation can significantly help reduce the overall overhead. For example, two AND gates are using the same inputs: $z_1 = x_1\, AND\, y$ and $z_2 = x_2\, AND\, y$. If executed without leveraging this observation, two multiplicative tuples $a_1 \times b_1 = c_1$ and $a_2 \times b_2 = c_2$ are needed, 4 secret bits, $x_1 - a_1, x_2 - a_2, y - b_1, y - b_2$ should be revealed. This will incur at least 4 bit traffic overhead. However, if we leverage this observation, we can use a composed multiplicative tuple $(a_1, a_2) \times b = (c_1, c_2)$ and open only 3 secret bits, $x_1 - a_1, x_2 - a_2, y - b$, to accomplish the computation.
In general, if there are n ANDs sharing the same input wire, we can save $n - 1$ bits out of the $2n$ bits of incurred traffic overhead.

Generating the composed tuples can be done similarly as outlined above. The key difference is when generating the product shares the messages sent through Oblivious Transfer will have bits equal to the composed tuple size, and the shared bit b will be the choice bit used in the Oblivious Transfer.

MPC Standard Library

We provided a number of standard libraries on top of the basic frontend types to partially mitigate the incompatibility of some standard library functions. However, due to the nature of the protocol, some implementations of those libraries have their own assumptions and security guarantees, while some other implementations can inherit the assumption and guarantee of the backend engine. For example, the oblivious RAM protocol from DS17 only works for two-party scenarios under the semi-honest assumption while a linear ORAM can work in any scenario supported by the backend engine.

Oblivious RAM

An *Oblivious Random Access Memory* (ORAM) allows securely accessing an array of secret values with a secret index. We have implemented two types of ORAM: Linear ORAM and Write-Only ORAM.

Each ORAM is initialized with the size of the array, and this size directly affects the efficiency of the ORAM. For both types of ORAM, we support the following operations. The method `obliviousAddBatch` takes as input secret indices and secret values, and computes running sums at each index by aggregating the secret values at the corresponding secret indices. The `publicRead` and `secretRead` methods can be used to obtain the running sum at a given index, in plaintext and in secret shares respectively.

Linear ORAM

The linear ORAM works as follows. To access the array at a given secret index *i*, the linear ORAM traverses all the available secret indices, and compares each secret index with *i*, where the comparison is done privately such that no information about the indices is revealed. If the index is equal to *i*, then we add a secret value to the running sum at that index. If not, then we perform a dummy operation by adding zero to the running sum. Although this algorithm seems rather naive and inefficient, it outperforms the next implementation when the array size is very small.

Write-Only ORAM

The write-only ORAM uses the protocol from DS17, which is a lot more complicated than the linear ORAM but performs better for larger array sizes. Currently we only implemented the write-only ORAM protocol in this paper since it's the most efficient one that meets our needs. In the future, we will add the implementation for other variants as well, as the write-only ORAM can only support the two-party semi-honest scenario. This protocol can only support the two-party semi-honest scenario.

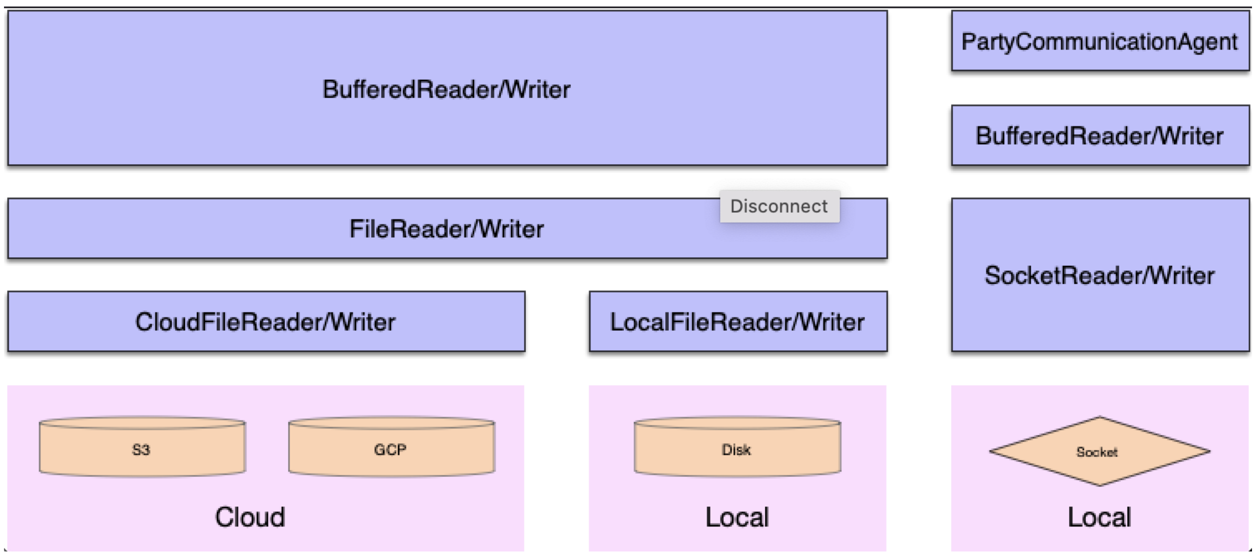
Oblivious Permute and Shuffle

Permutation is trivial in plaintext, but oblivious permute is quite difficult without revealing the permuted order. An oblivious permutation function is provided in PCF to cover this gap. The underlying algorithm is the AS-Waksman Network. Our permuter can permute a batch of values to a specific new order, provided by one of the parties. Shuffling can be realized by letting parties take turns to permute the data with their own randomness. Our oblivious permuter and shuffler inherits the security guarantee provided by the cryptographic backend.

I/O Systems and Security

The diagram below illustrates the different levels of abstraction that PCF I/O implements:

- On the bottom level, there are APIs for reading/writing from raw sinks - LocalFileReader/Writer, CloudFileReader/Writer, and SocketReader/Writer.
- One level above, there is FileReader/Writer where the location of the files can be determined by the framework.
- On the top level, there is BufferedReader/Writer that operates on large chunks of data to minimize the number of I/O operations needed with the underlying data source/sink.



Buffering

In PCF I/O, buffered reader and writer support will make it easier to upload and download "chunks" of data from either local storage or various cloud storage such as Amazon Web Services (AWS) S3 or Google Cloud Platform (GCP) Cloud Storage. This is desirable whenever you have a very large file to process that can't entirely fit in memory. By default, we load 4KB of data from the cloud source and then expose read() and readLine() methods to the user to provide a more developer-friendly interface.

Currently, AWS and GCP are the only supported sources for buffered I/O. S3 supports reading a range of bytes by setting the SetRange header in a GetObjectRequest call which internally will add a Range header to the HTTP request. GCS similarly supports a Range header to get a specific range of bytes, so the API in PCF I/O is equivalent between the two.

Buffered writing support is currently in progress, and there is an opportunity to utilize MultiPart uploads supported by both S3 and Cloud Storage. The developer can "chunk" uploads into multiple pieces to avoid loading an entire result into memory and then sending it over the network. One huge benefit here is that if the network fails, we only need to retry uploading the specific part instead of redoing the entire upload. Both S3 and Cloud Storage support uploading a maximum of 10,000 parts, which PCF I/O handles for you behind the scenes.

Socket Communication and TLS

PCF uses a basic TCP server/client model to communicate between parties. The SocketPartyCommunicationAgent is used by both parties to create either a client or a server port, and the IP/port of the server must be passed in by the client.

PCF is currently in the process of adding TLS support for communication between the two parties using OpenSSL. We plan to achieve mutual TLS authentication between parties so both sides can verify the identity of the other. OpenSSL offers off the shelf APIs for TCP sockets, like SSL_connect and SSL_accept that take the place of traditional POSIX APIs.

Future Development Opportunities

Arithmetic Circuit and Signal Conversion

In addition to the semi-honest XOR-secret-sharing-based MPC protocol, we are also implementing the similar MPC protocol for arithmetics. To solder the two protocols together, a signal conversion algorithm will be provided as well.

Arithmetic-Secret-Sharing-Based MPC

The Arithmetic Secret Sharing Protocol is an extension of the XOR based one which operates on numbers of a given modulus rather than boolean values (i.e. modulus 2). The protocol allows parties to calculate any arithmetic function over inputs that are provided by each party.

Similar to the XOR share model, all private values are split between the parties using secret shares which are held as integers in Z_n where n is the given modulus. To recover the value, the parties exchange the values and add them up in the modulus n. For example if $n = 10$, there are 3 parties, and the shares are $5, 1, 9$, the recovered secret shared value is $5 + 1 + 9 \, mod \, 10 = 5 \, mod \, 10$.

The operations supported by the protocol are Addition (PLUS), Multiplication (MULT), and Negation (NEG). For example, two parties knowing x and y could compute $f(x, y) = 5x^2 + 3xy + 4x + 3y \, mod \, n$ without having to reveal x and y to one another.

To compute the PLUS operation of two private values $u = u_A + u_B \, mod \, n$ and $v = v_A + v_B \, mod \, n$, the parties simply sum their shares locally with no communication. Thus $w_A = u_A + v_A \, mod \, n$ and $w_B = u_B + v_B \, mod \, n$. It is simple to show that w_a and w_b are shares of $w = u + v \, mod \, n$ since addition modulo n is commutative.

The NEG operation is a special case of multiplication by a public value (-1). Because we are in $mod \, n$, the parties will multiply their shares by $n - 1$. Given $u_A + u_B = u \, mod \, n$ are the secret shares of u held by the parties, then $w_A = n - u_A \, mod \, n$, $w_B = n - u_B \, mod \, n$ and $w_A + w_B =- u$.

To compute multiplication, the parties will again need to generate a one time multiplicative triple that can be used to secretly compute the result. A multiplicative triple is three numbers a, b, c in Z_n such that $a \ * \ b = c \mod n$. As always, the parties only know their shares of the values such that $(a_A + a_B) \ * (b_A + b_B) = (c_A + c_B) \mod n$.

The parties have secret shares of $u = u_A + u_B \mod n$ and $v = v_A + v_B \mod n$ and would like to compute shares w_A, w_B of $w = u * v \mod n$.

The parties will first reveal the values $u + a \mod n$ and $v + b \mod n$ by revealing the respective shares to each other. This costs $\log n$ bits of communication. With these values, the can locally compute their shares of w as follows:

i. $w_A = c_A - (u + a)b_A - a_A(v + b) + (u + a)(v + b) \mod n$
ii. $w_B = c_B - (u + a)b_B - a_B(v + b) \mod n$

Indeed, we see that

$$w_A + w_B = c_A + c_B - (u + a)(b_A + b_B) - (a_A + a_B)(v + b) + (u + a)(v + b) \mod n$$
$$w_A + w_B = c - (u + a)b - a(v + b) + (u + a)(v + b) \mod n$$
$$w_A + w_B = ab - ub - ab - av - ab + uv + ub + av + ab = uv \mod n$$

Signal Conversion Protocols

Being able to convert signals between XOR secret-sharing and arithmetic sharings of different modulars is the most critical functionality to support circuits of mixed types of computations. We will implement different signal conversion protocols from different research papers for various scenarios and underlying MPC protocols.

Oblivious Sort

In addition to ORAM, permuting, and shufflings, sorting is another important functionality for many applications. Our strategy is to implement a sorting network on top of MPC so that a set of values can be obliviously sorted.

Covert Security

All of our implemented components are targeting the semi-honest model. However, the soundness of this security model in the real-world is questionable, especially when the stake is significantly high. A stronger security model is the only solution here. A good candidate model that balances between performance and security is the covert security model. A potential next step is to implement a MPC protocol that can provide security guarantees in presence of a covert adversary.