# MP-SPDZ: A Versatile Framework for Multi-Party Computation

Marcel Keller
CSIRO's Data61
marcel.keller@data61.csiro.au
mks.keller@gmail.com

## ABSTRACT

MP-SPDZ is a fork of SPDZ-2 (Keller et al., CCS '13), an implementation of the multi-party computation (MPC) protocol called SPDZ (Damgård et al., Crypto '12). MP-SPDZ extends SPDZ-2 to more than twenty MPC protocols, all of which can be used with the same high-level programming interface based on Python. This considerably simplifies comparing the cost of different protocols and security models.

The protocols cover all commonly used security models (honest/dishonest majority and semi-honest/malicious corruption) as well as computation of binary and arithmetic circuits (the latter modulo primes and powers of two). The underlying primitives employed include secret sharing, oblivious transfer, homomorphic encryption, and garbled circuits.

The breadth of implemented protocols coupled with an accessible high-level interface makes it suitable to benchmark the cost of computation in various security models for researchers both with and without a background in secure computation

This paper aims the outline the variety of protocols implemented and the design choices made in the development of MP-SPDZ as well as the capabilities of the programming interface.

## 1 INTRODUCTION

Multi-party computation allows a set of parties to compute collaboratively on their private inputs without revealing anything other than the result. A range of applications have been suggested such as truthful auctions [BDJ+06], avoiding satellite collisions [HLOI16], computing the gender pay gap [LVB+16], or privacy-preserving machine learning [MZ17]. After the development of the basic theory in the 1980s [CCD88, BGW88], the first implementation in software was created by Pinkas et al. [PSSW09] in 2009. Since then, a number of frameworks have been created for a range of security models and computation domains. By security model we mean the assumptions on the behavior of the parties, in particular how many are assumed to behave honestly and whether dishonest parties follow the protocol or try to gain information or compromise the result by deviating. On the other hand, the computation domain denotes a mathematical structure that is used to represent secret information in the computation as required by the underlying schemes. In applications, this domain usually takes the form of a ring, that is, a set with the two operations akin to addition and multiplication. Often said ring is simply defined by integer operations with modulus, but there are other example such as the Galois fields of characteristic two (e.g., the one defining the arithmetic structure of AES).

Almost all openly available frameworks for multi-party computation are restricted to a particular security model and computation domain, which makes it hard to assess the cost of one security model compared to another because one has to an implement the same computation several times. Similarly, it is hard to compare protocols in the same security model. MP-SPDZ[1] aims to change this by providing an implementation of the more than twenty protocols with the same virtual machine, and a compiler that compiles high-level code into bytecode to be executed by said virtual machine. This allows to implement a computation only once before benchmarking it in a variety of settings.

At the core of the approach of combining many protocols in one framework lies the intuition that, despite the differences, all commonly used protocols for secure computation can largely be reduced to a few operations that are similar in all of them, and these are input, output, locally computable linear operations, one much more involved operation like AND or multiplication, and generating correlated randomness such as random bits. It therefore seems natural to build a general framework facilitating this blueprint in order to reuse components and optimizations as much as possible. MP-SPDZ does so while still allowing specific protocols to be added if necessary.

MP-SPDZ is based on SPDZ-2 [KRSS18, KSS13], an implementation of the SPDZ protocol [DPSZ12, DKL+13]. The infrastructure of SPDZ-2 has been used and extended in a line works [KS14, KOS15, KOS16, Kel17, KY18, KPR18, DEF+19]. Araki et al. [ABF+18] have undertaken steps to integrate protocols other than SPDZ in SPDZ-2. However, their approach is marred by overly relying on library calls for basic operations such as share additions, and thus it is relatively inefficient. This can be seen by how well SPDZ fares in comparison to simpler honest-majority protocols (Figures 6-8 by Araki et al.) because the former is natively implemented in SPDZ-2. MP-SPDZ is not based on the work by Araki et al. but directly forked from SPDZ-2 instead.

Hastings et al. [HHNZ19] have assessed eleven multi-party computation frameworks available in the first half of 2018. They estimate that creating and running three simple example programs in all of the frameworks together took them 750 person-hours. MP-SPDZ aims to lower the entry barrier to secure computation by providing standalone Linux binaries for every release and extensive documentation for the high-level language.[2] In addition, it uses a continuous integration facility, and issues filed on GitHub are usually attended within a few days.

*Paper organization.* After introducing similar frameworks and basic concepts, we will proceed layer by layer, from the underlying protocols to the high-level library. At the end, we will present benchmarks comparing MP-SPDZ to similar frameworks.

[1] https://github.com/data61/mp-spdz
[2] https://mp-spdz.readthedocs.org

As an anchor throughout the paper, we will use the example of computing an inner product of two vector supplied to by two different parties. Hastings et al. have used this as one of their examples. The code for MP-SPDZ looks as follows:

```
a = [sint.get_input_from(0) for i in range(n)]
b = [sint.get_input_from(1) for i in range(n)]
res = sint.dot_product(a, b)
print_ln('%s', res.reveal())
```

## 1.1 Comparison to Other Frameworks

In the following we will consider all frameworks described by Hastings et al. and the ones subsequently added to their repository containing example programs and build environments [HHNZ20] at the time of writing. Where available we also indicate the approximate size of the inner product example. Table 1 highlights how MP-SPDZ compares to other frameworks. SH stands for semi-honest security. See Section 8 for details.

*ABY.* This framework only implements two-party computation with semi-honest security [DSZ15]. Unlike MP-SPDZ it provides conversion between secret-sharing computation and garbled circuits, however. The inner product example takes about 60 lines of code without comments or empty lines.

*ABY$^3$.* This framework only implements three-party computation with honest majority and semi-honest security [MR18]. The inner product example takes about 40 lines of code without comments or empty lines.

*CBMC-GC.* This is a compiler that compiles C code to a binary circuit description [BHWK16] to be executed by ABY. Hastings et al. were unable to make it work with their examples.

*EMP-toolkit.* This framework only implements garbled circuits in various security models [WMK16]. If not using Yao's garbled circuits with semi-honest security, loops are unrolled at compile time. The inner product example takes about 60 lines of code without comments or empty lines.

*FRESCO.* This framework [Ale20] only implements dishonest majority computation, with malicious security for arithmetic circuits (SPDZ and SPD$\mathbb{Z}_{2^k}$) and with semi-honest security for binary circuits [DNNR17]. The inner product example takes about 30 lines of code without comments or empty lines.

*Frigate.* This is a compiler that compiles C-like code to a binary circuit description [MGC$^+$16]. Unlike with MP-SPDZ, loops are always unrolled at compile-time. The inner product example takes about 20 lines of code without comments or empty lines.

*JIFF.* This JavaScript library only implements honest-majority computation with semi-honest security as a whole [Tea20]. Unlike MP-SPDZ it allows to change the security model between offline and online phase. The inner product example takes about 50 lines of code without comments or empty lines.

*MPyC.* This Python framework [Sch20] only implements computation with semi-honest security based on Shamir's secret sharing [Sha79]. The inner product example takes about 20 lines of code without comments or empty lines.

*Obliv-C.* This framework compiles an extension of C to machine code via standard C [ZE15]. It only supports Yao's garbled circuits with semi-honest security. The inner product example takes about 20 lines of code without comments or empty lines.

*OblivVM.* This framework compiles an extension of Java to Java bytecode [LWN$^+$15]. It only supports Yao's garbled circuits with semi-honest security. The inner product example takes about 20 lines of code without comments or empty lines.

*PICCO.* This framework compiles an extension of C to native binaries via standard C [ZSB13]. It only implements semi-honest computation with an honest majority based on Shamir's secret sharing. The inner product example takes about 10 lines of code without comments or empty lines.

*SCALE-MAMBA.* This framework [COS19] is another fork of SPDZ-2 [KSS13, KRSS18]. Despite the common roots, the two forks have diverged considerably since 2018. SCALE-MAMBA only implements arithmetic computation modulo a prime (not modulo a power of two), garbled circuits according to Hazay et al. [HSS17], and binary computation based on secret sharing [FKOS15, WRK17]. All computations are implement with malicious security only, and dishonest majority computation modulo a prime is only implemented using homomorphic encryption. On the other hand, SCALE-MAMBA implements honest-majority computation for any access structure possible in theory.

The frontend is similar to the one in MP-SPDZ but without later additions such as the dynamic loop optimization (Section 6.3), the repetitive code optimization (Section 6.4), and machine learning functionality (Section 7.3). Furthermore, the authors have started to move away from the Python compiler in favor of a new one based on Rust.

The inner product example takes less than 10 lines of code without comments or empty lines.

*Sharemind MPC.* This framework implements a frontend for a variety of backends but its own backend only uses three-party semi-honest computation with an honest majority [BLW08]. It also allows to use ABY and FRESCO as backend while the proprietary backend is not freely available. The inner product example takes less than 10 lines of code without comments or empty lines.

*TinyGarble.* This framework only implements Yao's garbled circuits with semi-honest security [SHS$^+$15]. Hastings et al. were unable to make it work with their examples.

*Wysteria.* This framework implements a domain-specific language with only binary computation in the semi-honest setting with dishonest majority [RHH14]. Hastings et al. were unable to run all their examples in this framework.

# 2 PRELIMINARIES

In this section we explain various basic concepts in secure computation.

*Security models.* A core property of multi-party computation protocols is what assumptions are made about the parties, which happens along two axes. The first question is how many parties are assume to behave honestly and how many are assumed to

| | SH 2-party (OT) | SH replicated 3-party | Shamir | Malicious Shamir | SPDZ (HighGear) | Yao's GC (32-bit) |
|---|---|---|---|---|---|---|
| ABY | 1.8 | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| ABY$^3$ | ⊥ | 0.02 | ⊥ | ⊥ | ⊥ | ⊥ |
| EMP-toolkit | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | 93* |
| MPyC | ⊥ | ⊥ | 13.21 | ⊥ | ⊥ | ⊥ |
| Obliv-C | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | 29 |
| ObliVM | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | 700 |
| PICCO | ⊥ | ⊥ | 0.10 | ⊥ | ⊥ | ⊥ |
| SCALE-MAMBA | ⊥ | ⊥ | ⊥ | 9.9 | 324 | ⊥ |
| MP-SPDZ | 0.8 | 0.01/0.04 | 0.06/0.11 | 0.4 | 33 | 59 |

**Table 1: Time in seconds to compute an inner product of two 100,000-element vectors of 64-bit integers.**

be "corrupted" in some way. The protocols in MP-SPDZ all assume threshold corruption, that is a constant number of parties are thought to be corrupted. There is a considerable difference between an honest majority and a dishonest majority, that is, whether this threshold is strictly below half the number of parties or not. The protocols for a dishonest majority are generally more intricate and more expensive.

The second question is how corrupted parties behave. The two main categories here are whether they still follow the protocol but collude on extracting information (called semi-honest or passive corruption) or whether they also deviate from the protocol in order to gain information or distort the result of the computation (called malicious or active corruption). The latter setting raises further questions as to whether the computation can still continue in case of deviation (guaranteed output) and whether the corrupted parties can learn the result without honest parties doing so (fairness). All protocols in MP-SPDZ with malicious security work in the malicious-with-abort model, which means that deviation is detected but the protocol cannot recover after that. The reason is that fairness is relatively expensive, and guaranteed output is outright impossible with dishonest majority. Consider the case of a two-party protocol: One party cannot continue when the other stops communicating.

*Computation domains.* There is a variety of mathematical structures used in multi-party computation. Often this takes the form of integers modulo some number $M$. If $M = 2$ the computation reduces to binary circuits because addition then corresponds to XOR and multiplication corresponds to AND. For larger $M$, the literature has established the term arithmetic circuits because the basic operations provided remain addition and multiplication. If $M$ is a prime, the domain satisfies the definition of a field, that is, all numbers except zero have a multiplicative inverse. This is required for a number schemes including Shamir's secret sharing [Sha79] and SPDZ [DPSZ12]. For the latter, these requirements were later adapted for computation modulo a power $2^k$, resulting in the SPD$\mathbb{Z}_{2^k}$ protocol. However, integer computation modulo a prime is not the only example of a finite field. MP-SPDZ also implements computation in $\mathbb{F}_{2^k}$. The latter domain particularly has applications for symmetric cryptosystems such as AES, which is based on arithmetic in $\mathbb{F}_{2^8}$.

*Secret sharing.* This concept of distributing information is at the core of many multi-party computation protocols. Information is distributed in a way that allows some sets (called qualified) of parties to reconstruct it while some smaller sets (called unqualified) cannot deduce anything from the shares they are given. The exact definition of which subsets can and cannot recover is called an access structure. Clearly, a superset of a qualified has to be qualified as well, and a subset of an unqualified set has be to unqualified for the access structure to make sense. Furthermore, the common definition of secret sharing requires that all subsets of parties are either qualified or unqualified.

As an example consider additive secret sharing. A number $x$ in a range $[0, M-1]$ is shared among parties $P_1, \ldots, P_n$ by sending a random number $x_i \in [0, M-1]$ to $P_i$ such that $\sum_i x_i = x \mod M$. It is easily seen that the all parties together can reconstruct $x$ while the view of any strict subset is indistinguishable to a random set of numbers. This secret sharing scheme is used in all protocols with dishonest majority except schemes based on garbling.

Other secret sharing schemes used in MP-SPDZ include replicated secret sharing [BL90] where one starts with additive secret sharing but sends more than one share to every party and Shamir's secret sharing [Sha79] where shares are determined using a random polynomial.

All of these secret sharing schemes are linear, that is, any (affine) linear combination of secret-shared values can be computed locally because the reconstruction is linear as well.

*Beaver's multiplication.* This technique [Bea92] reduces multiplication of secret numbers to a multiplication of secret random numbers, which is useful both in terms of security and practicality. First, it allows to execute the multiplication of secret random numbers optimistically before checking whether the parties followed the protocol. If the check fails the protocol can abort without consequences because the secret randomness is independent of the actual secret data. Second, Beaver's multiplication allows to execute the preprocessing batch-wise even if the actual computation is sequential, which is a particular advantage when using lattice-based cryptosystems that (only) allow to encrypt many values at once efficiently.

The multiplication works as follows: Let denote $[x]$ a secret sharing of $x$, and assume that a triple $([a], [b], [ab])$ for independent random $a$ and $b$ is available. The multiplication of $[x]$ and $[y]$ can then be computed after revealing $x + a$ and $y + b$. Revealing these

does not reveal $x$ or $y$ because $a$ and $b$ are secret and randomly chosen. After revealing the sums, the parties compute

$$[xy] = [(x + a - a) \cdot (y + b - b)] \\ = (x + b) \cdot (y + b) - (x + a) \cdot [b] - (y + b) \cdot [a] + [ab].$$

Note that this computation is affine linear with respect to the secret values and thus it is possible to compute it with any linear secret sharing scheme.

*Oblivious transfer.* This is a basic two-party cryptographic protocol that was the first to be used for dishonest-majority computation. Essentially, one party (called sender) inputs two strings $s_0, s_1$, and the other party (called receiver) inputs a bit $b$ and learns $s_b$. Crucially, the sender does not learn $b$, and the receiver does not learn $s_{1-b}$. It is relatively straightforward to construct a protocol from oblivious transfer that allows two parties to compute a secret sharing of the product of private numbers known to each without revealing the inputs. The construction is leveraged by protocols such as MASCOT [KOS16]. Oblivious transfer is only known to be constructed from public-key cryptography. However, OT extension [IKNP03] can be used keep the use of such more expensive schemes at a low constant while using more symmetric cryptography instead.

*Homomorphic encryption.* This refers to encryption schemes that allow to operate on ciphertexts efficiently in a way that implies some operation without revealing the cleartext. While encryption schemes with limited homomorphism have been known for a while (textbook RSA is homomorphic in some sense), only the emergence of lattice-based cryptography led to scheme that are homomorphic with respect to two operations. MP-SPDZ uses the leveled encryption scheme by Brakerski et al. [BGV12], which requires that a vector of numbers are encrypted at once. The minimum length depends on the plaintext modulus but generally ranges from several thousand to several ten thousand. The concrete implementation is based on the one by Gentry et al. [GHS12], which is defined by the use of Montgomery representation [Mon85] and fast Fourier transform. The first allows to compute multiplication modulo a prime without expensive modular reduction, and the second as an efficient method for converting a list of numbers to a polynomial in the plaintext space such that the multiplication of two such polynomial corresponds to an element-wise product of the numbers. This approach imposes a restriction on the prime modulus that can be used, but many applications requiring only general integer-like computation are indifferent to the actual prime.

## 3 THE PROTOCOLS

Table 2 shows how many protocol variants in each combination of security model and computation domain are implemented in MP-SPDZ. In the following we will briefly describe them.

### 3.1 Dishonest Majority

All protocols for dishonest majority use techniques related to public-key cryptography, either oblivious transfer of homomorphic encryption. Research on practical protocols in this line was started by Nielsen et al. [NNOB12] and Damgård et al. [DPSZ12]. The former proposed a two-party protocol computing binary circuits based on oblivious transfer (later dubbed TinyOT), and the latter proposed a multi-party protocol for computation in fields (modulo a prime or $\mathbb{F}_{2^k}$) using homomorphic encryption (named SPDZ after the authors). Both use Beaver's technique of dividing the computation in a data-independent and a data-dependent phase [Bea92]. The former (sometimes called "offline") outputs correlated information that is used by latter together with the actual private data.

*MASCOT.* This protocol denotes a way of computing Beaver triples used in SPDZ using oblivious transfer (OT) with malicious security [KOS16], which was the first offline phase released in SPDZ-2. A core technology here is the use of so-called OT extension with malicious security [KOS15], which considerably increases the throughput because public-key cryptography is only used briefly at the beginning of the computation.

*SPD$\mathbb{Z}_{2^k}$.* MASCOT has been adapted to computation modulo a power of two [CDE+18, DEF+19]. The main challenge here is that not every element in $\mathbb{Z}_{2^k}$ has an inverse, which is a crucial tool in proving the security of MASCOT (and SPDZ). SPD$\mathbb{Z}_{2^k}$ mitigates this by moving to $\mathbb{Z}_{2^{k'}}$ for a larger $k'$ in order to counter the effect of zero divisors. MP-SPDZ fully implements SPD$\mathbb{Z}_{2^k}$ using its own implementation of $\mathbb{Z}_{2^k}$ optimized with compile-time $k$.

*SPDZ and Overdrive.* LowGear and HighGear [KPR18] are the names of two protocols computing Beaver triples for SPDZ using semi-homomorphic and somewhat homomorphic encryption, respectively. They were part of SPDZ-2. Semi-homomorphic means that it is possible to add ciphertexts and multiply ciphertexts with cleartexts in order to obtain an encryption of the sum and the product, respectively. By somewhat homomorphic we mean that in addition it is also possible to multiply two ciphertexts but the result cannot be further multiplied by a ciphertext. All known homomorphic cryptosystems incur a considerable performance penalty for every additional level of ciphertext multiplication. The core idea of SPDZ, LowGear, and HighGear is to minimize the number of such multiplication levels using multi-party computation techniques. LowGear and HighGear represent a trade-off in that LowGear runs a sub-protocol between all pairs of parties (similar to Bendlin et al. [BDOZ11]) and therefore does not scale as well with the number of parties as HighGear.

Since MP-SPDZ does not implement the key generation needed for LowGear and HighGear with malicious security, variants of them using covertly secure key generation account for the two covertly secure protocols in Table 2. Optionally, the TopGear zero-knowledge proof [BCS19] can be used instead of the original variants.

MP-SPDZ also retains the offline phase of earlier variants of the SPDZ protocol, with malicious [DPSZ12] and covert security [DKL+13]. Since they are superseded by HighGear, they are not integrated with the online phase and therefore do not appear in Table 2, however.

*Binary secret sharing.* MP-SPDZ supports two ways of computing binary circuits with secret sharing, dishonest majority, and malicious security. The first is adapted from SPD$\mathbb{Z}_{2^k}$ with $k = 1$. However, this variant suffers from the fact that communication is quadratic in the security parameters. Even though that also holds for SPD$\mathbb{Z}_{2^k}$ but has less of impact if $k$ is larger than the security

| Security model | Modulo prime / Galois field | Modulo power of two | Binary sharing | Garbling |
|---|---|---|---|---|
| Malicious, dishonest majority | 1 | 1 | 2 | 1 |
| Covert, dishonest majority | 2 | 0 | 0 | 0 |
| Semi-honest, dishonest majority | 3 | 1 | 1 | 2 |
| Malicious, honest majority | 3 | 4 | 2 | 1 |
| Semi-honest, honest majority | 2 | 1 | 2 | 1 |

**Table 2: Number of protocols implemented in MP-SPDZ.**

parameter as is the case for SPD$\mathbb{Z}_{2^k}$ because the communication of protocols based on oblivious transfer is in $O(k^2)$.

The second variant is based on a multi-party extension of TinyOT [FKOS15] and has communication linear in the security parameter.

*BMR.* Beaver et al. [BMR90] have presented a way to construct garbled circuits from any multi-party computation scheme inheriting the security properties. Their approach was later refined by Lindell et al. [LPSY15] by using SPDZ as the underlying protocol. MP-SPDZ implements BMR using SPDZ/MASCOT as well as protocols in other security models. This functionality was never part of SPDZ-2 but has been partially published before the first version of MP-SPDZ because it was used by Keller and Yanai to implement oblivious RAM [KY18].

*Yao's Garbled Circuits.* Bellare et al. [BHKR13] have presented a variant of Yao's garbled circuits that works particularly well with AES-NI, the native implementation of AES on modern processors. An implementation was added after the last version of SPDZ-2.

*Semi-honest security.* It is relatively straight-forward to convert a protocol with malicious security to one with semi-honest security by removing all aspects that solely contribute to malicious security. All protocols in the line of SPDZ and TinyOT feature two main elements for this: an information-theoretic tag (called "MAC" in most of the literature) and a procedure called sacrificing that checks the correctness of correlated randomness using more correlated randomness. The latter has to be discarded thereafter because the procedure reveals correlation between the two parts, hence the name sacrificing. Stripping both leaves a protocol that comes close to canonical semi-honest protocol using the underlying techniques (oblivious transfer or homomorphic encryption). The main difference is that in the semi-honest setting one could use OT or HE directly with secret data instead of produced correlated randomness. Given the cost of OT or HE, the overhead is relatively small compared to the additional of Beaver's multiplication, however. Furthermore, using correlated randomness with homomorphic encryption can easily work with the SIMD structure of LWE-based schemes, that is the fact that such schemes only can encrypt thousands of field elements at once for reasonable security parameters. This allows to use a single encryption even for computing sequential multiplications.

As a result, MP-SPDZ implements an OT-based protocol in all computation domains as well protocols for fields both with semi-homomorphic and somewhat homomorphic encryption.

## 3.2 Honest Majority

The honest-majority setting allows to compute securely entirely using secret sharing, without oblivious transfer or homomorphic encryption. MP-SPDZ uses two secret sharing schemes to this end, replicated secret sharing [BL90] and Shamir's secret sharing [Sha79]. Both are multiplicative, that is, it is possible to locally compute a sharing of the multiplication of two sharings without communication, albeit resulting in a different secret sharing scheme. A resharing protocol involving communication can then convert back to the original sharing scheme, thus allowing further multiplication.

The resharing protocol is linear in that resharing of a sum is equivalent to the sum of resharings, which allows to compute an inner product at the communication cost of a single multiplication. The framework reflects this by providing a particular interface for inner products as seen later.

*Semi-honest computation based on replicated secret sharing.* Araki et al. [AFL+16] have observed that resharing can be done by every party sending exactly one element of the computation domain by using pseudo-random zero sharing [CDI05]. The latter refers to the generation of a fresh random sharing of zero without communication using pseudo-random number generation with a set of keys shared between every pair of parties. A similar technique allows to reduce the communication for private inputs as noted by Eerikson et al. [EKO+19].

*Semi-honest computation based on Shamir's secret sharing.* This goes back to Ben-Or et al. [BGW88] for computation module a prime and Chaum et al. [CCD88] for computation in extension fields of characteristic two (and thus binary circuits). MP-SPDZ implements resharing as described by Cramer et al. [CDM00].

*Malicious computation.* Lindell et al. [LN17] have adapted the SPDZ sacrifice to the setting of replicated secret sharing modulo a prime, and Araki et al. [FLNW17] have done so for TinyOT. It is straight-forward to do the same with Shamir secret sharing. For computation modulo a power of two, Eerikson et al. [EKO+19] have presented several variants, one more akin to TinyOT, another inspired by the SPD$\mathbb{Z}_{2^k}$ sacrifice, and a third called post-sacrifice. The last was introduced by Lindell et al. [LN17] for computation modulo a prime, and it works by executing any multiplication with only semi-honest security but storing the inputs and output for a later check with a random triple similar to a SPDZ sacrifice.

### 3.3 Higher-Level Protocols

All protocols described above are generally concerned with an implementation of an arithmetic black-box, that is private input, addition, multiplication, and public output. However, many computations such as comparison require further correlated randomness, most notably secret random bits in the larger domains. A simple way of obtaining such bits is to have sufficiently many parties (depending on the security model) input a random bit and then computing the XOR of all of them. However, this does not scale optimally with the number of parties because computing XOR in a larger domain reduces to a multiplication: $a \oplus b = a + b - 2ab$. Furthermore, for malicious security, it has to be checked whether the output actually is a bit.

For computation modulo a prime, Damgård et al. [DKL+13] have shown that a secret random bit can be produced at the cost of only one multiplication and one opening by using the fact that the square of a random number does not reveal which of two possible square roots it corresponds to (if not zero). Damgård et al. [DEF+19] later extended this to computation modulo a power of two.

*Arithmetic-binary conversion.* MP-SPDZ implements several ways of converting between arithmetic (modulo a larger number) and binary (modulo two) computation. The general way uses correlated randomness in the two domains, so-called doubly-authenticated bits (daBits) introduced by Rotaru and Wood [RW19], and extended daBits introduced by Escudero et al. [EGK+20]. In semi-honest computation with a low number of parties, there is also the possibility of a more direct conversion [ABF+18, MR18].

## 4 INTERNAL INTERFACES

In multi-party computation based on secret sharing it is essential that parallel communication is bundled up to some extent. This raises the question how to translate that requirement into an interface that is as easy to use as possible. Consider the following simple operator overloading:

```
a = b * c
e = d * f
```

The two multiplications can be run in parallel. However, the communication implied by the first line has to be deferred at least until the execution of the second line, which means that that `a` cannot represent an actual share but a deferred value instead, which in turn requires a considerable machinery in the background to handle the the communication and the availability of deferred values. Such an approach has been taken by VIFF [Gei07], its successor MPyC [Sch20], and FRESCO [Ale20]. In contrast, MP-SPDZ aims for more efficiency at this stage and defers usability to a higher level. Therefore, the internal interface requires the programmer indicate parallel computation. A strictly vector-based interface is one way of doing this, but MP-SPDZ offers a slightly more amenable variant. Assume that `a` and `b` are vectors of multiplicands of size `n` and `c` is supposed to hold the output.

```
protocol.init_mul();
for (int i = 0; i < n; i++)
  protocol.prepare_mul(a[i], b[i]);
protocol.exchange();
for (int i = 0; i < n; i++)
  c[i] = protocol.finalize_mul();
```

This interface allows a more dynamic use without the necessity of copying information into vectors first.

As every protocol offers the same interface, it is straightforward to implement the same computation for several protocols at once using C++ templating also because the input and output protocols use the same approach.

Coming back to our inner product example, Figure 1 shows its implementation using the internal interface of an arithmetic protocol. As the communication takes place in the `exchange` calls, it is parallelized as much as possible. Furthermore, the code uses the optimized dot product facility if available for the protocol.

```
for (int i = 0; i < n; i++)
  input.input_from_all();
input.exchange();
for (int i = 0; i < n; i++)
{
  a[i] = input.finalize(0);
  b[i] = input.finalize(1);
}
protocol.init_dotprod();
for (int i = 0; i < n; i++)
  protocol.prepare_dotprod(a[i], b[i]);
protocol.next_dotprod();
protocol.exchange();
for (int i = 0; i < n; i++)
  c = protocol.finalize_dotprod();
output.prepare_open(c);
output.exchange();
result = output.finalize_open();
```

**Figure 1: Inner product computation with C++ in MP-SPDZ.**

### 4.1 Templating

C++ templating is widely used in MP-SPDZ because it allows to reuse code whenever suitable without performance penalty. If runtime polymorphism with virtual functions would be used instead, every additional protocol would incur extra cost at every point of the execution where protocols differ. Given that the protocols implemented in the framework differ in very small units (e.g., adding shares can involve as little as adding two of 64-bit numbers but also adding several pairs of numbers modulo a large prime), we estimate that the cost of runtime polymorphism would be considerable.

As an example, the implementation of Beaver's multiplication [Bea92] requires only about 100 lines of code while being used for over ten protocols spanning all computation domains and several security models.

The central class of any protocol represents a secret value in that protocol. Everything else like types for corresponding cleartext values and sub-protocols are derived from this class using `typedef`. Usually classes describing a particular protocol have one template variable for the secret value class, from which all derived information can be accessed.

## 4.2 Preprocessing

A considerable part of the backend is dedicated to the so-called preprocessing, which denotes the generation of correlated randomness. The most prominent example of such randomness are multiplication triples $([a], [b], [ab])$ where $a$ and $b$ are uniformly random and all numbers are secret-shared. Such triples are central to the Beaver's multiplication. Other examples include squares $(a, a^2)$, inverses $(a, a^{-1})$, random bits, and (e)daBits [EGK+20].

MP-SPDZ provides implementations to generate preprocessing generically for semi-honest and malicious security as well as specific implementations for certain protocols. For example, it is more efficient to generate squares without using multiplication in SPDZ [DKL+13], or daBits can be generated from random bits in some protocols.

Since the marginal cost per preprocessed element is lower if they are generated in batches, the framework provides infrastructure to do so. Furthermore, it is possible to read such information from disk, which allows to benchmark only the phase of the computation that is dependent on secret data (sometimes called "online phase").

Preprocessing is generally executed on demand to avoid unnecessary computation, which is different to SCALE-MAMBA [COS19] where even the lesser used squares are generated whenever a computation is run. This does not only slow down the computation, it can also cause the application to seemingly hang at the end because the preprocessing has not finished.

# 5 THE VIRTUAL MACHINE

Keller et al. [KSS13] have presented a virtual machine designed for the specifics of multi-party computation. This virtual machine is at the core of SPDZ-2 [KRSS18], the predecessor of MP-SPDZ. The main characteristic of the virtual machine is that instructions involving communication allow an unrestricted number of arguments and thus minimizing the number of communication rounds. It is this property that distinguishes it from many other virtual machines and processors. Consider for example the 64-bit x86 instruction set. While instructions vary in the number of clock cycles required for completion (as little as one for addition but more than 100 for sine or cosine [F+11]), the difference between instructions is easily quantified by the difference in the binary circuits required to compute them. In multi-party computation, there is not only a quantitative difference between an addition of shares and a multiplication but also a qualitative one because the former can be done locally while the latter involves communication. This qualitative difference has implications that vary from protocol to protocol, but the benefit of unrestricted parallelization of communication is straightforward due to network latencies.

In the context of garbled circuits, there is another benefit of the parallelization property. Bellare et al. [BHKR13] proposed implementing garbled circuits using AES-NI, the CPU-native implementation of AES. Their scheme makes use of so-called pipelining, which means that several AES operations can be run in parallel on the same CPU core if the instructions are executed directly one after the other. The virtual machine design allows to make use of pipelining while relieving the user in this respect.

*High-level design.* SPDZ-2 implements the SPDZ online phase for computations in $\mathbb{F}_p$ for a large prime $p$ and $\mathbb{F}_{2^n}$ with either $n = 40$ or $n = 128$. MP-SPDZ adds a range of protocols, replacing $\mathbb{F}_p$ with $\mathbb{Z}_{2^k}$ for some protocols and adding computations of Boolean circuits (technically in $\mathbb{F}_2$). The latter is rooted in the implementation used by Keller and Yanai [KY18], and it works with vectors of length 64 by default. This is necessary for efficiency with some protocols where a share only consists of a bit or a pair of bits, and it leads to a natural optimization by using the 64-bit machine words of contemporary processors. As a result, every instantiation of the virtual machine offers integer-like computation (in $\mathbb{F}_p$ or $\mathbb{Z}_{2^k}$), and computation for $\mathbb{F}_{2^n}$ and Boolean circuits, all in the same security model.

*Basic data types.* The virtual machine allows to handle data of public and secret values for every computation domain and 64-bit integers. The reason for having clear data types in every domain on top of another integer type is that the size of numbers in the computation domains vary and numbers in $\mathbb{F}_p$ are stored using Montgomery representation, which does not lend itself for purposes such as loop counters or addressing memory. It is therefore cleaner to have public data types reflecting all computation domains. Furthermore, in the context of garbled circuits there is a conceptual difference between public numbers that are known prior to garbling such as loop counters and numbers that result from revealing a secret value. This difference matters because the garbling part of a computation depends on a revealed value has to happen after the evaluation resulting in said value whereas garbling only depending on a loop counter can be processed at any time.

*Registers.* The virtual machine provides an unlimited number of registers for every basic data type. While this is less sophisticated than a stack-based design, it allows for a simpler implementation. Register numbers are hard-coded into the bytecode, which allows the virtual machine to allocate sufficient numbers for the computation. Registers are generally used as to store inputs and outputs of instructions, and they are local to a thread.

*Memory.* For more complex data structures such as arrays, matrices, and higher-dimensional structures, the virtual machine provides another facility for every basic data type, called the memory. The memory arrays are global, and thus allow to communicate information between threads. Unlike registers, the memory can be accessed using runtime values stored in integer registers. Memory has to be allocated at compile time.

*Instructions.* Most of the instructions supported by the virtual machine can be roughly categorized as follows:

**Copying** This includes initializing registers, copying between registers and memory, as well as conversion between registers of different public data types.

**Simple computation** Instructions resembling the common three-argument format is used for all computation that does not require communication, including linear operations on secret values.

**Secure protocols** All instructions of this category allow an infinite number of arguments to facilitate reducing communication rounds as described above. The protocols include essential ones such as private input, multiplication, and public output, but also more specialized ones such as inner product,

matrix multiplication, and the special truncation by Dalskov et al. [DEK19].

**Preprocessed information** As mentioned in Section 4.2, preprocessing is executed batch-wise. This means that instructions of this type only incur communication if necessary. It is also possible to communicate the amount of preprocessed information required in a tape in order to optimize the preprocessing.

**Control flow** The virtual machine allows jumps as well as spawning and joining threads.

**Further input/output** Instructions in this category facilitate functionality outside secure protocols such as printing or client communication.

**Protocol information** It is possible to use the same program in various player configurations by accessing information such as the number of players and then for example loop over all available players when gathering inputs.

*Vectorization.* Most instructions are vectorized, that is, they imply the execution of the same operation for as many consecutive registers as requested. This considerably reduces the overhead of representing repetitive computation both during compilation as well as during execution. The virtual machine also facilitates structured loading of values into consecutive registers, for example, loading rows or columns at any dimension in multi-dimensional arrays.

*Threads.* The virtual machine implements multi-threading as follows: A computation is run in the main thread as described by list of instructions called a "tape". The latter can start further tapes in other threads and wait for their completion. However, the maximum number of threads has to be known at compile time. Despite this limitation, the available functionality is powerful enough for many applications that benefit from multi-threading such as matrix multiplication or convolutions.

*Inner product example.* Figure 2 shows the bytecode representation of our inner product example with vectors of length three. Note that both `inputmixed`, `dotprods`, `asm_open` give the number of parameters as first argument. The first instruction lets private inputs from parties 0 and 1 to be stored in registers `s1,s3,s5` and `s2,s4,s6`, respectively, and the result of the inner product is then stored in `s0`. Eventually, the inner product is revealed and printed followed by a new line character.

# 6 THE COMPILER

Similar to SPDZ-2, the compiler runs high-level code written in Python, and outputs bytecode to be run by the virtual machine. Some aspects have been changed as pointed out throughout this chapter. The following core aspects remain the same, however.

*Type system.* MP-SPDZ follows the dynamic typing paradigm of Python. This makes programming more intuitive in that, for example, any operation involving a secret and a public value results in secret value. Consider that in a stronger type system, the assignment of a secret type to a public type would involve automatic revealing, which can be unintended, or the compiler would need to produce an error, which makes programming harder. Dynamic typing together with a clearly named way of revealing secret information therefore strikes a compromise between security and ease of use.

*Basic blocks.* This concept is taken from general compiler design and denotes a sequence of instructions without branching. In our context, the compiler performs the round-minimizing optimization only in the context of a basic block because it requires rearranging instructions.

## 6.1 Minimizing the Number of Rounds

This is the main optimization conducted by the compiler. It analyses a basic block to find instructions of the same kind that can be merged into a single instruction because they are independent. Recall that such instructions allow an infinite number of arguments. An easy example of a possible merge are two multiplications that can be run in parallel. MP-SPDZ differs from SPDZ-2 in that it merges different operations independently while SPDZ-2 reduces multiplication to openings using Beaver multiplications. Such a reduction clearly excludes protocols that do not use Beaver multiplications. As a result, MP-SPDZ uses an instruction for secure multiplication, which is merged independently of opening operations. Another difference to SPDZ-2 is that the latter splits the opening process in two instructions called `startopen` and `stopopen`, which allows to execute local computation while waiting for information from the network. However, this comes at the cost of an increased complexity in the compiler as well as the fact to the number of parallel opens is limited by the communication buffer, which has to be considered on the compiler level. MP-SPDZ uses a single instruction for opening, which simplifies the handling.

The algorithm proceeds in three steps:

(1) First it creates a dependency graph of all instructions in a basic block. This graph takes the form of an directed acyclic graph where the nodes stand for instructions. There are a few reasons to create an edge between two nodes, most notably if the output of one instruction is an input to another. Other dependencies include various types of interaction with the environment, e.g., the order of input read from a party should not change.
(2) The algorithm assigns instructions that can be merged to rounds. All instructions in the same round have the same type. This is done by a variant of the longest-path algorithm:
    (a) All nodes without predecessors are assigned to round zero.
    (b) Every non-mergeable node is assigned the maximum of the round number of its predecessors.
    (c) Every mergeable node is assigned the minimal round that is larger than all of its predecessors and that is compatible, that is, it is not occupied by an instruction of another type.

    Since this algorithm only consider predecessors of nodes, it can be run at the same time as the dependency graph creation.
(3) All instructions in the same round are merged. This involves merging the arguments such that the semantics are preserved, and merging all edges in the dependency graph.
(4) The instructions are output in topological order according to the changed dependency graph.

```
inputmixed 18, 0, s1, 0, 0, s3, 0, 0, s5, 0, 0, s2, 1, 0, s4, 1, 0, s6, 1 # 0
dotprods 8, 8, s0, s1, s2, s3, s4, s5, s6 # 1
asm_open 2, c0, s0 # 2
print_reg_plain c0 # 3
print_char 10 # 4
```

**Figure 2: Virtual machine code of our inner product example with vectors of length three.**

*Memory instruction dependency.* Since memory instructions allow runtime addresses, it is not straightforward how treat them in the dependency graph. If every read instruction is made to depend on a write instruction and vice versa, a possibly large potential for minimizing rounds is lost. Consider an unrolled loop where a memory address is read followed by some computation on the read value, the result of which is stored at the same address, and the same is repeated for more addresses. All computations can clearly be executed in parallel, but a dependency of every memory instruction on the previous one prevents this. The compiler therefore only considers dependencies if they involve the same address, be it a compile-time address or the same register for run-time addresses. In combination with caching registers when accessing data structures such as a multi-dimensional arrays, this strikes a balance between efficiency and correctness guarantees. Furthermore, the compiler offers a command to start a new basic block, which inherently preserves the order of instructions.

*Dead code elimination.* The dependency graph created above also allows to eliminate instructions with unused results. An instruction is considered obsolete if it is not considered inherently essential (because of side-effects on the environment or the memory) and if all successors are obsolete. A simple backward pass suffices to determine and eliminate obsolete instructions.

## 6.2 Register Allocation

As in SPDZ-2, the compiler initially uses an unlimited supply of write-once registers, which are allocated to a minimal number of registers at the end. For a straightline program without branches, this is trivial by passing through backwards, allocating a register whenever it is read for the last time, and deallocating it at the single time it is written to. For a program with branches there is the difficulty that some registers have to be live throughout a loop to prevent overwrites after the last read. This is solved by allocating registers that are written to before the loop starts before processing the parts of the loop.

## 6.3 Loops

While the nature of multi-party computation makes it non-trivial to implement loops that depend on secret data, loops depending on public data naturally reduce the representation of computation. Similar to SPDZ-2, MP-SPDZ supports loops depending both on compile-time and run-time public data. The former still allows compile-time analysis of the computation cost. In the high-level language loops can be executed using function decorators as follows:

```
@for_range(n)
def _(i):
    a[i] = ...
```

While this is certainly unusual, it allows compilation by running the high-level Python code within the scope of the compiler. Furthermore, it enables a variety of loops without creating a domain-specific language. For example, while `@for_range` creates a strict run-time loop executed consecutively, `@for_range_opt` implements the dynamic optimization described below, `@for_range_multithread` and `@for_range_opt_multithread` execute a loop in a fixed number of threads.

*Dynamic loop optimization.* Büscher et al. [BDK+18] have described a trade-off between unrolling loops in order to merge communication rounds as in Section 6.1 and limiting memory usage during compilation caused by the increased space to represent the unrolled computation. They propose to use a dynamic approach by unrolling until a time budget is exhausted. MP-SPDZ adapts their approach by using a budget on the number of instructions produced.

## 6.4 Repetitive Code

Since the basic computation in multi-party computation is limited to additions and multiplications in a domain, even seemingly simple computations such as comparisons translate into a non-trivial composition of basic operations. The design principle of MP-SPDZ is to break down computation into basic operations in order to limit the complexity of the virtual machine. However, this means that repetitive code leads to a repetitive expansion of the same building block. To avoid that cost, the compiler offers to treat certain basic computations atomically and merging them as in Section 6.1. While speeding up compilation this has the downside that some parallel computations will be executed sequentially. For example, a comparison and an equality test on independent data can be executed in parallel (thus reducing the number of rounds), but treating the two separately will execute them sequentially. Because of the trade-off, this optimization is only used when requested.

Examples of building blocks treated in the way above range from integer operations such as comparison and truncation to mathematical functions such as the trigonometric functions.

# 7 HIGH-LEVEL LIBRARY

In this section, we will describe the high-level library, that is, the capabilities for secret computation implemented on top of the compiler. These implementations generally involve breaking down the desired functionality to the basic operations supported by the protocols such as input, output, and arithmetic in the computation domain. As these operation are modeled by the arithmetic black-box,

**Pre:** $[x]$ for $x \in \mathbb{Z}_{2^k}$, $[r_0], \ldots, [r_{k-1}]$ such that $r_i \xleftarrow{\$} \{0,1\}$
**Post:** $[x_0], \ldots, [x_{k-1}]$ such that $x = \sum_i x_i \cdot 2^i$ and $x_i \in \{0,1\}$

(1) Compute and open $[c] = [x] + \sum_i [r_i] \cdot 2^i$
(2) Compute $c_0, \ldots, c_{k-1}$ such that $c = \sum_i c_i \cdot 2^i$ and $c_i \in \{0,1\}$
(3) Compute $[x_0], \ldots [x_{k-1}]$ using a binary adder on $([r_0], \ldots, [r_{k-1}])$ and $(c_0, \ldots, c_{k-1})$

**Figure 3: Bit decomposition modulo $2^k$ using random bits.**

it is usually straightforward to prove the security of the extended functionality.

## 7.1 Integer Operations

Beyond basic arithmetic in the respective domain, the library implements comparison, equality, left/right shift, modulo power of two, and power of two with secret exponent. A core component of most of these operations in larger computation domains is a "mask-and-open" approach. This involves adding or subtracting a secret random value of a certain form to or from a secret value and opening the result, which can then be processed as a public value, for example to extract single bits. See Figure 3 for the basic example of a bit decomposition. It uses these public bits together with the secret bits of the masking allows to compute a bit decomposition of a secret value, that is, an individual secret sharing for every bit instead of a secret sharing for the whole value. Unlike the latter, the former allows to compute all integer operations above directly. Catrina and de Hoogh [Cd10] have shown how to optimize this simple approach with computation modulo a prime, and later that was adapted by Dalskov et al. [DEK19] for computation modulo a power of two, and by Escudero et al. [EGK+20] for switching to binary computation for some operations.

Mask-and-open with computation modulo a prime is only statistically secure and requires the secret value to be within an assumed interval. For example, assume that $x \in [0, 2^l]$ and that $r \in [0, 2^{l+s}]$ is randomly chosen for a security parameter $s$. Then, $x + r$ is statistically indistinguishable from a uniform number in $[0, 2^{l+s}]$. In computation modulo a power of two this is not an issue because overflow bits can simply be erased by multiplying with a power of two.

## 7.2 Fractional Numbers

As SPDZ-2, MP-SPDZ offers two ways of representing fractional numbers, fixed-point and floating-point. The former denotes representing a fractional number $x$ by an integer $y$ such that $x = y \cdot 2^{-f}$ for a fixed precision $f$. The latter is similar to floating-point representations such as IEEE 754 in that $x$ is represented by the four-tuple $(v, p, z, s)$ such that

$$x = \begin{cases} (-1)^s \cdot v \cdot 2^p & z = 0 \\ 0 & z = 1. \end{cases}$$

The additional bits $s$ and $z$ simplify the computation given that secure computation does not directly allow to access single bits of a larger value.

*Fixed-point numbers.* Due to the larger efficiency, this is the preferred approach for fractional numbers in MP-SPDZ. Addition and subtraction are straight-forward by the linearity of the representation, and multiplication corresponds to integer multiplication followed by truncation of $f$ bits. The truncation can either be computed as a left shift or a more efficient probabilistic truncation. The latter involves randomized rounding based on the input. For example, 0.25 would be rounded to 0 with probability 0.25 and to 1 with probability 0.75 when rounding to integers. Catrina and Saxena first suggested this for computation modulo a prime [CS10], Dalskov et al. [DEK19] adapted it for computation modulo a power of two, and Escudero et al. [EGK+20] to mixed computation. Catrina and Saxena also presented how to use Goldschmidt's algorithm [Gol64] to compute division in secure computation.

*Floating-point numbers.* Aliasgari et al. [ABZS13] have shown how to implement floating-point numbers in the context of secure computation modulo a prime. Their approach translates directly to $\mathbb{Z}_{2^k}$ with the exception of multiplying by $(2^m)^{-1}$ in order to compute a left shift for $x$ when $x = 0 \mod 2^m$. This can be achieved through bit decomposition, however. The library implements addition, subtraction, division, and comparisons of floating-point numbers. Note that the approach by Aliasgari et al. is not fully compliant to IEEE754.

*Mathematical functions.* Aly and Smart [AS19] have implemented a number of mathematical functions in secure computation, ranging from trigonometric functions over square root to exponential and logarithm. MP-SPDZ integrates this by reusing code provided by the authors through SCALE-MAMBA [COS19]. All functions are implemented for fixed-point numbers while the implementations for floating-point numbers are restricted to sine, cosine, and tangent.

## 7.3 Further Functionality

The library extends beyond basic mathematics as shown in the following.

*Machine learning.* The library provides functionality for logistic regression [KS19] and deep-learning inference [DEK19]. The former allows to choose between an accurate implementation of the sigmoid function and an approximation similar to ABY$^3$ [MR18]. The latter is restricted to the quantization scheme used in MobileNets [JKC+17].

*Oblivious data structures.* MP-SPDZ retains the code used by Keller and Scholl [KS14]. This includes an oblivious array, queue, and stack implementation. Oblivious here means that all accesses are done secretly, that is, without revealing indices (where applicable) and whether an access is reading or writing. However, it is inherent to secure computation that an upper limit to the amount of data in the structure is revealed. Oblivious RAM [SvS+13] is the core technique to achieve efficient data structures for larger sizes. Based on the data structures above, the library contains example implementations of Dijkstra's algorithm for shortest path in graphs, and the Gale-Shapley algorithm for stable matching.

*Binary circuits.* The library allows to process binary circuits in the so-called Bristol Fashion format. This text-based format has

been established by the authors of SCALE-MAMBA [COS19], inspired by similar formats used for other secure computation frameworks. SCALE-MAMBA comes a selection of example circuits such as the Keccak sponge function [BDPVA09]. Based on the latter, the library provides the computation of SHA3 for short inputs. Unlike SCALE-MAMBA, MP-SPDZ processes binary circuits in the compiler in the compiler in order to optimize them for secret sharing computation and Yao's garbled circuits. SCALE-MAMBA only uses Bristol Fashion circuits in the context of BMR-style garbled circuits where such optimizations matters less because the key used in AES is not fixed.

## 8 BENCHMARKS

We have benchmarked MP-SPDZ against other frameworks that support arithmetic circuits. Table 3 (repeating Table 1) shows how many seconds it takes to compute an inner product of length 100,000 on one machine with 7-th generation i7 processor (baseline frequency 2.8 GHz). We have opted for a local computation because not all frameworks support execution on different hosts.

We have replaced `get_input_from()` by `get_raw_input_from()` in the MP-SPDZ code for benchmarking. This is because the former leads to inputs being read using the `istream` functionality of the C++ standard library, which considerably increases the time in some benchmarks. The examples in other frameworks either use faster functions like `atoi()` or the inputs are hard-coded. Furthermore, the timings for computation modulo a prime (Shamir and SPDZ) have been done using a 128-bit prime because that is a common choice given that many algorithms require a $(k + s)$-bit prime for $k$-bit computation and security parameters $s$. We have reduced all statistical security parameters in SCALE-MAMBA to 40 for a fairer comparison. Finally, some benchmarks for MP-SPDZ show two times, the pure computation time and for the overall time including parsing the program description.

The table shows that the more sophisticated approach using a virtual machine does not degrade performance considerably with arithmetic computation.

In addition, we have benchmarked frameworks using Yao's garbled circuits but with 32-bit integers instead of 64 because some examples only support the former. Table 3 shows that only Obliv-C is considerably faster because it implements an optimization that MP-SPDZ does not, namely not garbling AND gates if either input is known to one of the parties. The result for EMP-toolkit is marked with an asterisk because it is actually ten times the result for an inner product of size 10,000. We did not manage to run an inner product of size 100,000.

Note that $\perp$ means that a particular framework does not implement a particular protocol, except for the case of ABY and Yao where there is no example code available.

*Missing frameworks.* We do not provide benchmarks for various frameworks mentioned in Section 1.1 for the following reasons:

- No working inner product example: CBMC-GC, TinyGarble, Wysteria
- No backend executing the actual computation: Frigate
- Inner product example incomplete (without preprocessing): FRESCO
- Using JavaScript: JIFF
- Software not available: Sharemind MPC

## REFERENCES


[ABF+18] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, and Hikaru Tsuchida. Generalizing the SPDZ compiler for other protocols. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 880–895. ACM Press, October 2018.

[ABZS13] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. In *NDSS 2013*. The Internet Society, February 2013.

[AFL+16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, October 2016.

[Ale20] Alexandra Institute. FRESCO - a FRamework for Efficient Secure COmputation, 2020. https://github.com/aicis/fresco.

[AS19] Abdelrahaman Aly and Nigel P. Smart. Benchmarking privacy preserving scientific operations. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19*, volume 11464 of *LNCS*, pages 509–529. Springer, Heidelberg, June 2019.

[BCS19] Carsten Baum, Daniele Cozzo, and Nigel P. Smart. Using TopGear in overdrive: A more efficient ZKPoK for SPDZ. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019*, volume 11959 of *LNCS*, pages 274–302. Springer, Heidelberg, August 2019.

[BDJ+06] Peter Bogetoft, Ivan Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In Giovanni Di Crescenzo and Avi Rubin, editors, *FC 2006*, volume 4107 of *LNCS*, pages 142–147. Springer, Heidelberg, February / March 2006.

[BDK+18] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 847–861. ACM Press, October 2018.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Heidelberg, May 2011.

[BDPVA09] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3(30):320–337, 2009.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*, pages 478–492. IEEE Computer Society Press, May 2013.

[BHWK16] Niklas Büscher, Andreas Holzer, Alina Weber, and Stefan Katzenbeisser. Compiling low depth circuits for practical secure computation. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *ESORICS 2016, Part II*, volume 9879 of *LNCS*, pages 80–98. Springer, Heidelberg, September 2016.

[BL90] Josh Cohen Benaloh and Jerry Leichter. Generalized secret sharing and monotone functions. In Shafi Goldwasser, editor, *CRYPTO'88*, volume 403 of *LNCS*, pages 27–35. Springer, Heidelberg, August 1990.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206. Springer, Heidelberg, October 2008.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th ACM STOC*, pages 11–19. ACM Press, May 1988.

| | SH 2-party (OT) | SH replicated 3-party | Shamir | Malicious Shamir | SPDZ (HighGear) | Yao's GC (32-bit) |
|---|---|---|---|---|---|---|
| ABY | 1.8 | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| ABY$^3$ | ⊥ | 0.02 | ⊥ | ⊥ | ⊥ | ⊥ |
| EMP-toolkit | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | 93* |
| MPyC | ⊥ | ⊥ | 13.21 | ⊥ | ⊥ | ⊥ |
| Obliv-C | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | 29 |
| ObliVM | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | 700 |
| PICCO | ⊥ | ⊥ | 0.10 | ⊥ | ⊥ | ⊥ |
| SCALE-MAMBA | ⊥ | ⊥ | ⊥ | 9.9 | 324 | ⊥ |
| MP-SPDZ | 0.8 | 0.01/0.04 | 0.06/0.11 | 0.4 | 33 | 59 |

**Table 3: Time in seconds to compute an inner product of two 100,000-element vectors of 64-bit integers.**


[Cd10] Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 182–199. Springer, Heidelberg, September 2010.

[CDE+18] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod $2^k$ for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Heidelberg, August 2018.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 342–362. Springer, Heidelberg, February 2005.

[CDM00] Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 316–334. Springer, Heidelberg, May 2000.

[COS19] KU Leuven COSIC. SCALE-MAMBA, 2019. https://github.com/KULeuven-COSIC/SCALE-MAMBA.

[CS10] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 35–50. Springer, Heidelberg, January 2010.

[DEF+19] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy*, pages 1102–1120. IEEE Computer Society Press, May 2019.

[DEK19] Anders Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. Cryptology ePrint Archive, Report 2019/131, 2019. https://eprint.iacr.org/2019/131.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013.

[DNNR17] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 167–187. Springer, Heidelberg, August 2017.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015.

[EGK+20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. Cryptology ePrint Archive, Report 2020/338, 2020. https://eprint.iacr.org/2020/338.

[EKO+19] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use your brain! Arithmetic 3PC for any modulus with active security. Cryptology ePrint Archive, Report 2019/164, 2019. https://eprint.iacr.org/2019/164.

[F+11] Agner Fog et al. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering*, 93:110, 2011.

[FKOS15] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 711–735. Springer, Heidelberg, November / December 2015.

[FLNW17] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 225–255. Springer, Heidelberg, April / May 2017.

[Gei07] Martin Geisler. VIFF: Virtual ideal functionality framework. http://viff.dk/, 2007.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 850–867. Springer, Heidelberg, August 2012.

[Gol64] Robert E. Goldschmidt. Applications of division by convergence. Master's thesis, MIT, 1964.

[HHNZ19] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: General purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy*, pages 1220–1237. IEEE Computer Society Press, May 2019.

[HHNZ20] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. Sample code and build environments for mpc frameworks, 2020. https://github.com/MPC-SoK/frameworks.

[HLOI16] Brett Hemenway, Steve Lu, Rafail Ostrovsky, and William Welser IV. High-precision secure computation of satellite collision probabilities. In Vassilis Zikas and Roberto De Prisco, editors, *SCN 16*, volume 9841 of *LNCS*, pages 169–187. Springer, Heidelberg, August / September 2016.

[HSS17] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 598–628. Springer, Heidelberg, December 2017.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[JKC+17] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, abs/1712.05877, 2017.

[Kel17] Marcel Keller. The oblivious machine - or: How to put the C into MPC. In Tanja Lange and Orr Dunkelman, editors, *LATINCRYPT 2017*, volume 11368 of *LNCS*, pages 271–288. Springer, Heidelberg, September 2017.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Heidelberg, August 2015.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Heidelberg, April / May 2018.

[KRSS18] Marcel Keller, Dragos Rotaru, Peter Scholl, and Nigel P. Smart. Multiparty computation with SPDZ, MASCOT, and Overdrive offline phases, 2018.

https://github.com/bristolcrypto/SPDZ-2.

[KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 506–525. Springer, Heidelberg, December 2014.

[KS19] Marcel Keller and Ke Sun. A note on our submission to track 4 of iDASH 2019. Cryptology ePrint Archive, Report 2019/1246, 2019. https://eprint.iacr.org/2019/1246.

[KSS13] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 549–560. ACM Press, November 2013.

[KY18] Marcel Keller and Avishay Yanai. Efficient maliciously secure multiparty computation for RAM. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 91–124. Springer, Heidelberg, April / May 2018.

[LN17] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 259–276. ACM Press, October / November 2017.

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 319–338. Springer, Heidelberg, August 2015.

[LVB+16] Andrei Lapets, Nikolaj Volgushev, Azer Bestavros, Frederick Jansen, and Mayank Varia. Secure mpc for analytics as a web application. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 73–74. IEEE, 2016.

[LWN+15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376. IEEE Computer Society Press, May 2015.

[MGC+16] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 112–127, 2016.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[MR18] Payman Mohassel and Peter Rindal. ABY$^3$: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, October 2018.

[MZ17] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38. IEEE Computer Society Press, May 2017.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, Heidelberg, December 2009.

[RHH14] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy*, pages 655–670. IEEE Computer Society Press, May 2014.

[RW19] Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 227–249. Springer, Heidelberg, December 2019.

[Sch20] Barry Schoenmakers. MPyC: Secure multiparty computation in Python. https://github.com/lschoe/mpyc, 2020.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[SHS+15] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE Computer Society Press, May 2015.

[SvS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, November 2013.

[Tea20] Multiparty.org Development Team. JavaScript implementation of federated functionalities, 2020. https://github.com/multiparty/jiff.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[WRK17] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 39–56. ACM Press, October / November 2017.

[ZE15] Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. http://eprint.iacr.org/2015/1153.

[ZSB13] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: a general-purpose compiler for private distributed computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 813–826. ACM Press, November 2013.

# A INNER PRODUCT EXAMPLES

## A.1 ABY

```
/**
 \file                innerproduct.h
 \author         sreeram.sadasivam@cased.de
 \copyright        ABY - A Framework for Efficient Mixed-protocol Secure Two-party Computation
                    Copyright (C) 2015 Engineering Cryptographic Protocols Group, TU Darmstadt
                    This program is free software: you can redistribute it and/or modify
                    it under the terms of the GNU Affero General Public License as published
                    by the Free Software Foundation, either version 3 of the License, or
                    (at your option) any later version.
                    This program is distributed in the hope that it will be useful,
                    but WITHOUT ANY WARRANTY; without even the implied warranty of
                    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
                    GNU Affero General Public License for more details.
                    You should have received a copy of the GNU Affero General Public License
                    along with this program. If not, see <http://www.gnu.org/licenses/>.
 \brief               Implementation of the Inner Product using ABY Framework.
 */

#ifndef __INNERPRODUCT_H_
#define __INNERPRODUCT_H_

#include "../../../abycore/circuit/booleancircuits.h"
#include "../../../abycore/circuit/arithmeticcircuits.h"
#include "../../../abycore/circuit/circuit.h"
#include "../../../abycore/aby/abyparty.h"
#include "../../../abycore/sharing/sharing.h"
#include <math.h>
#include <cassert>

using namespace std;

/**
 \param               role                role played by the program which can be server or client part.
 \param         address         IP Address
 \param         seclvl                Security level
 \param         nvals               Number of values
 \param         bitlen                Bit length of the inputs
 \param         nthreads        Number of threads
 \param               mt_alg               The algorithm for generation of multiplication triples
 \param         sharing               Sharing type object
 \param         num                       the number of elements in the inner product
 \brief               This function is used for running a testing environment for solving the
 Inner Product.
 */
int32_t test_inner_product_circuit(e_role role, char* address, uint16_t port, seclvl seclvl,
               uint32_t nvals, uint32_t bitlen, uint32_t nthreads, e_mt_gen_alg mt_alg,
               e_sharing sharing, uint32_t num);

/**
 \param               s_x                       share of X values
 \param               s_y                share of Y values
 \param         num                       the number of elements in the inner product
 \param               ac                        Arithmetic Circuit object.
 \brief               This function is used to build and solve the Inner Product modulo 2^16. It computes the inner product
```

```
                        multiplying each value in x and y, and adding those multiplied results to evaluate the inner
                        product. The addition is performed in a tree, thus with logarithmic depth.
 */
share* BuildInnerProductCircuit(share *s_x, share *s_y, uint32_t num, ArithmeticCircuit *ac);


#endif

/**
 \file                innerproduct.cpp
 \author         sreeram.sadasivam@cased.de, heavily modified by marcella
 \copyright        ABY - A Framework for Efficient Mixed-protocol Secure Two-party Computation
 Copyright (C) 2015 Engineering Cryptographic Protocols Group, TU Darmstadt
 This program is free software: you can redistribute it and/or modify
 it under the terms of the GNU Affero General Public License as published
 by the Free Software Foundation, either version 3 of the License, or
 (at your option) any later version.
 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 GNU Affero General Public License for more details.
 You should have received a copy of the GNU Affero General Public License
 along with this program. If not, see <http://www.gnu.org/licenses/>.
 \brief               Implementation of the Inner Product using ABY Framework.
 */

#include "innerproduct.h"

int32_t test_inner_product_circuit(e_role role, char* address, uint16_t port, seclvl seclvl,
                uint32_t nvals, uint32_t bitlen, uint32_t nthreads, e_mt_gen_alg mt_alg,
                e_sharing sharing, uint32_t num) {

        /**
         Step 1: Create the ABYParty object which defines the basis of all the
         operations which are happening.        Operations performed are on the
         basis of the role played by this object.
         */
        ABYParty* party = new ABYParty(role, address, port, seclvl, bitlen, nthreads,
                        mt_alg);

        /**
         Step 2: Get to know all the sharing types available in the program.
         */
        vector<Sharing*>& sharings = party->GetSharings();

        /**
         Step 3: Create the circuit object on the basis of the sharing type
         being inputed.
         */
        ArithmeticCircuit* circ =
                        (ArithmeticCircuit*) sharings[sharing]->GetCircuitBuildRoutine();

        /**
         Step 4: Creating the share objects - s_x_vec, s_y_vec which
         are used as inputs to the computation. Also, s_out which stores the output.
         */
```

```
    share *s_x_vec, *s_y_vec, *s_out;

    /**
     Step 5: Allocate the vectors that will hold the plaintext values.
     */
    uint16_t x, y;

    uint16_t output;

    uint16_t * input = (uint16_t*) malloc(num * sizeof(uint16_t));

    uint32_t i;

    /**
     Step 6: Fill the input array with data read from file.
     Copy the input values into a share object for each party.
     The values for the party different from role is ignored,
     but PutINGate() must always be called for both roles.
     */
char *fname = (char *) malloc(100);
sprintf(fname, "/root/ABY/src/examples/innerprod/data/innerprod.%d.dat", role);

std::ifstream infile;
infile.open(fname);

for(i = 0; i < num; i++) {
    infile >> input[i] ;
    if(infile.eof()) {break;}
}

infile.close();

    s_x_vec = circ->PutSIMDINGate(i, input, 16, SERVER);
    s_y_vec = circ->PutSIMDINGate(i, input, 16, CLIENT);

    /**
     Step 7: Build the circuit, passing the input shares and circuit object.
     */
    s_out = BuildInnerProductCircuit(s_x_vec, s_y_vec, num, circ);

    /**
     Step 8: Output the value of s_out (the computation result) to both parties
     */
    s_out = circ->PutOUTGate(s_out, ALL);

    /**
     Step 9: Executing the circuit using the ABYParty object evaluate the
     problem.
     */
    party->ExecCircuit();

    /**
     Step 10: Type caste the plaintext output to 16 bit unsigned integer.
     */
    output = s_out->get_clear_value<uint16_t>();

    cout << "Circuit Result: " << output << endl;
```

```
        delete s_x_vec;
        delete s_y_vec;
    delete input;
        delete party;

        return 0;
}

/*
 Constructs the inner product circuit. num multiplications and num additions.
 */
share* BuildInnerProductCircuit(share *s_x, share *s_y, uint32_t num, ArithmeticCircuit *ac) {
        uint32_t i;

        // pairwise multiplication of all input values
        s_x = ac->PutMULGate(s_x, s_y);

        // split SIMD gate to separate wires (size many)
        s_x = ac->PutSplitterGate(s_x);

        // add up the individual multiplication results and store result on wire 0
        // in arithmetic sharing ADD is for free, and does not add circuit depth, thus simple sequential adding
        for (i = 1; i < num; i++) {
                s_x->set_wire_id(0, ac->PutADDGate(s_x->get_wire_id(0), s_x->get_wire_id(i)));
        }

        // discard all wires, except the addition result
        s_x->set_bitlength(1);

        return s_x;
}
```

## A.2 ABY$^3$

```
#include<iostream>
#include <cryptoTools/Network/IOService.h>

#include "aby3/sh3/Sh3Runtime.h"
#include "aby3/sh3/Sh3Encryptor.h"
#include "aby3/sh3/Sh3Evaluator.h"

#include "innerprod.h"

using namespace oc;
using namespace aby3;

void innerprod_test(oc::u64 partyIdx, std::vector<int>values) {
        if (partyIdx == 0)
            std::cout << "testing innerprod..." << std::endl;

        IOService ios;
        Sh3Encryptor enc;
        Sh3Evaluator eval;
        Sh3Runtime runtime;
        setup_samples(partyIdx, ios, enc, eval, runtime);
```

```
        // encrypt (only parties 0,1 provide input)
        u64 rows = values.size();
        si64Matrix A(1, rows);
        si64Matrix B(rows, 1);

        if (partyIdx == 0) {
            i64Matrix input(1, rows);
            for (unsigned i = 0; i < rows; i++)
              input(0, i) = values[i];
            enc.localIntMatrix(runtime, input, A).get();
        } else {
            enc.remoteIntMatrix(runtime, A).get();
        }

        if (partyIdx == 1) {
        i64Matrix input(rows, 1);
        for (unsigned i = 0; i < rows; i++)
            input(i, 0) = values[i];
        enc.localIntMatrix(runtime, input, B).get();
        } else {
            enc.remoteIntMatrix(runtime, B).get();
        }

        // parallel multiplications
        si64Matrix sum(1, 1);
        Sh3Task task = runtime.noDependencies();
        task = eval.asyncMul(task, A, B, sum);
        task.get();

        // reveal result
        i64Matrix result;
        enc.revealAll(runtime, sum, result).get();
        std::cout << "result: " << result(0, 0) << std::endl;
}
```

### A.3 CBMC-GC

```
// computes an inner product
//
// input: arrays of equal length
// output: integer inner product
//


#define LEN 10

typedef struct {
    int xs[LEN];
} Array;


int mpc_main(Array INPUT_A, Array INPUT_B) {

    int product = 0;

    for( int i=0; i<LEN; i++) {
        product += INPUT_A.xs[i] * INPUT_B.xs[i];
```

```
    }

    return product;

}
```

## A.4 EMP-toolkit

```
#include "emp-sh2pc/emp-sh2pc.h"
#include <new>
using namespace emp;
using namespace std;

int LEN = 10;

void test_innerprod(int bitsize, string inputs_a[], string inputs_b[], int len) {

    Integer sum(bitsize, 0, PUBLIC);
    Integer prod(bitsize, 0, PUBLIC);
        Integer a[len];
        Integer b[len];

    for( int i=0; i<len; i++) {
        a[i] = Integer(bitsize, inputs_a[i], ALICE);
        b[i] = Integer(bitsize, inputs_b[i], BOB);
    }

    for( int i=0; i<len; i++) {
        prod = a[i] * b[i];
        sum = sum + prod;
    }

    cout << "SUM: " << sum.reveal<int>() << endl;
}


int main(int argc, char** argv) {
    int bitsize;

    // generate circuit for use in malicious library
    if (argc == 2 && strcmp(argv[1], "-m") == 0 ) {
        setup_plain_prot(true, "innerprod.circuit.txt");
        bitsize = 16;
        string inputs[LEN] = {"0","0","0","0","0","0","0","0","0","0"};
        test_innerprod(bitsize, inputs, inputs, LEN);
        finalize_plain_prot();
        return 0;
    }

    // run computation with semi-honest model
    int port, party;
    parse_party_and_port(argv, &party, &port);
    NetIO * io = new NetIO(party==ALICE ? nullptr : "127.0.0.1", port);

    setup_semi_honest(io, party);

    if (argc != 4) {
```

```
      cout << "Usage: ./innerprod <party> <port> <bitsize>" << endl
           << endl;
      delete io;
      return 0;
    }

    cout << "Calculating inner product of two inputs of length " << LEN << endl;

    bitsize = atoi(argv[3]);

    char fname_a[40];
    char fname_b[40];

    sprintf(fname_a, "../data/innerprod/%d.1.dat", bitsize);
    sprintf(fname_b, "../data/innerprod/%d.2.dat", bitsize);

    ifstream infile_a(fname_a);
    ifstream infile_b(fname_b);

    string inputs_a[LEN];
    string inputs_b[LEN];

    if( infile_a.is_open() && infile_b.is_open()) {
        for( int i=0; i<LEN; i++) {
            getline( infile_a, inputs_a[i]);
            getline( infile_b, inputs_b[i]);
        }
        infile_a.close();
        infile_b.close();
    }

    test_innerprod(bitsize, inputs_a, inputs_b, LEN);
    delete io;
}
```

## A.5 Fresco

```
package dk.alexandra.fresco.samples.innerproduct;

import dk.alexandra.fresco.framework.Application;
import dk.alexandra.fresco.framework.DRes;
import dk.alexandra.fresco.framework.builder.numeric.ProtocolBuilderNumeric;
import dk.alexandra.fresco.framework.value.SInt;
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

/**
   Defines the MPC computation to input private values, compute the inner
   product, and output the result to all parties.
 */
public class InnerProduct implements Application<Integer, ProtocolBuilderNumeric> {

  private final List<BigInteger> vector;

  /**
```

```
     Constructs a new MPC computation for inner product using a given list of
     integers as the vector of this party.
   */
  public InnerProduct(List<Integer> vector) {
    this.vector = vector.stream().map(BigInteger::valueOf).collect(Collectors.toList());
  }

  @Override
  public DRes<Integer> buildComputation(ProtocolBuilderNumeric builder) {
    List<DRes<SInt>> sVec1 = new ArrayList<>(vector.size());
    List<DRes<SInt>> sVec2 = new ArrayList<>(vector.size());
    for (int i = 0; i < vector.size(); i++) {
      // Note: the below is a bit of a cheat as we are inputting the same vector
      // for both party 1 and 2. This works because if we are not party i our
      // input for party i will be disregarded.
      sVec1.add(builder.numeric().input(vector.get(i), 1));
      sVec2.add(builder.numeric().input(vector.get(i), 2));
    }
    DRes<SInt> result = builder.advancedNumeric().innerProduct(sVec1, sVec2);
    DRes<BigInteger> openResult = builder.numeric().open(result);
    return () -> openResult.out().intValue();
  }

}
```

*License.*

```
MIT License

Copyright (c) 2018 Security Lab // Alexandra Institute

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

## A.6 Frigate

```
/*
 * compute inner product of two vectors
 */

#define LEN 10
#parties 2

/* define some types */
```

```
typedef int_t 32 int
typedef uint_t 6 sint

typedef struct_t array
{
    int data[LEN];
}

/* define computation parameters */
#input 1 array
#output 1 int
#input 2 array
#output 2 int


function void main() {

    array xinput = input1;
    array yinput = input2;

    int result = 0;

    for(sint i=0; i<LEN; i++) {
        result = result + (xinput.data[i] * yinput.data[i]);
    }

    output1 = result;
    output2 = result;

}
```

### A.7 JIFF

```
(function (exports, node) {
  var saved_instance;

  /**
   * Connect to the server and initialize the jiff instance
   */
  exports.connect = function (hostname, computation_id, options) {
    var opt = Object.assign({}, options);
    // Added options goes here

    if (node) {
      // eslint-disable-next-line no-undef
      jiff = require('../../lib/jiff-client');
      // eslint-disable-next-line no-undef,no-global-assign
      $ = require('jquery-deferred');
    }

    // eslint-disable-next-line no-undef
    saved_instance = jiff.make_jiff(hostname, computation_id, opt);
    // if you need any extensions, put them here

    return saved_instance;
  };
```

```
  /**
   * The MPC computation
   */
  exports.compute = function (input, jiff_instance) {
    try {
      if (jiff_instance == null) {
        jiff_instance = saved_instance;
      }

      var final_deferred = $.Deferred();
      var final_promise = final_deferred.promise();

      // Share the arrays
      jiff_instance.share_array(input, input.length).then(function (shares) {
        try {
          // multipy all shared input arrays element wise
          var array = shares[1];
          for (var p = 2; p <= jiff_instance.party_count; p++) {
            for (var i = 0; i < array.length; i++) {
              array[i] = array[i].smult(shares[p][i]);
            }
          }

          // sum up elements
          var sum = array[0];
          for (var i = 1; i < array.length; i++) {
            sum = sum.sadd(array[i]);
          }

          // Open the array
          jiff_instance.open(sum).then(function (results) {
            final_deferred.resolve(results);
          });

        } catch (err) {
          console.log(err);
        }
      });

    } catch (err) {
      console.log(err);
    }

    return final_promise;
  };
}((typeof exports === 'undefined' ? this.mpc = {} : exports), typeof exports !== 'undefined'));
```

## A.8 MPyC

```
import numpy as np
from mpyc.runtime import mpc

async def main():

    # initialize mpc, define secure int type
    LEN = 100000
    await mpc.start()
```

```
    secint = mpc.SecInt(64)

    # initialize inputs
    values = [np.random.randint(1,1000) for _ in range(LEN)]

    n = len(mpc.parties)
    inputs = mpc.input([secint(v)for v in values], senders=list(range(n)))

    # compute pairwise products
    prod = inputs[0]
    for inp in inputs[1:]:
        prod = mpc.schur_prod(prod, inp)
    ip = mpc.sum(prod)

    # output result
    result = await mpc.output(ip)
    print("result:", result)
    await mpc.shutdown()


if __name__ == '__main__':

    mpc.run(main())
```

### A.9 Obliv-C

```
#include<obliv.oh>

#include"innerProd.h"

void dotProd(void *args){
  protocolIO *io = args;
  int v1Size = ocBroadcastInt(io->input.size, 1);
  int v2Size = ocBroadcastInt(io->input.size, 2);

  obliv int* v1 = malloc(sizeof(obliv int) * v1Size);
  obliv int* v2 = malloc(sizeof(obliv int) * v2Size);

  feedOblivIntArray(v1, io->input.arr, v1Size, 1);
  feedOblivIntArray(v2, io->input.arr, v2Size, 2);

  int vMinSize = v1Size<v2Size?v1Size:v2Size;

  obliv int sum = 0;
  for(int i=0; i<vMinSize; i++){
    sum += v1[i]*v2[i];
  }

  revealOblivInt(&(io->result), sum, 0);

}
```

### A.10 ObliVM

```
package com.github.danxinnoble.oblivm_benchmarker.innerProd;

int main@n@m(int@n x, int@m y){
  secure int32[public (n/32)] alc;
```

```
  secure int32[public (m/32)] bb;
  public int32 N = n/32;
  public int32 M = m/32;

  for(public int32 i=0; i<N; i=i+1){
    alc[i] = x$32*i~32*(i+1)$;
  }

  for(public int32 i=0; i<M; i=i+1){
    bb[i] = y$32*i~32*(i+1)$;
  }

  public int min;
  if(N < M){
    min=N;
  } else {
    min=M;
  }

  secure int32 sum = 0;
  for(public int32 i=0; i<min; i=i+1){
    sum = sum + (alc[i] * bb[i]);
  }

  return sum;
}
```

## A.11 PICCO

```
/*
 * takes the inner product of two inputs
 */

public int LEN = 10;

public int main() {

    int A[LEN], B[LEN];

    smcinput(A,1,LEN);
    smcinput(B,2,LEN);

    int p = A @ B;

    smcoutput(p,1);
    return 0;

}
```

## A.12 SCALE-MAMBA

```
sum = sint(0)
aaaas = sint.Array(10)
bbbbs = sint.Array(10)

for i in range(10):
  aaaas[i] = sint(i) # sint.get_private_input_from(0)
  bbbbs[i] = sint(i*2) # sint.get_private_input_from(1)
```

```
for i in range(10):
  sum = sum + (aaaas[i] * bbbbs[i])

print_ln("InnerProd: %s", sum.reveal())
```

## A.13 Sharemind

```
/*
 * Sharemind MPC example programs
 * Copyright (C) 2018 Sharemind
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <https://www.gnu.org/licenses/>.
 **/

import stdlib;
import shared3p;

domain pd_shared3p shared3p;


void main() {
        pd_shared3p uint64 [[1]] a = argument("a");
        pd_shared3p uint64 [[1]] b = argument("b");

        pd_shared3p uint64 c = sum(a * b);

        publish("c", c);
}
```

## A.14 License

The work by Hastings et al. in this section comes with the following license:

```
MIT License

Copyright (c) 2018-2019 Marcella Hastings, Brett Hemenway, Daniel Noble, Steve Zdancewic

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
```