Generalizing the SPDZ Compiler For Other Protocols[*]

Toshinori Araki[†] Assi Barak[‡] Jun Furukawa[§] Marcel Keller[¶] Yehuda Lindell[‡]
Kazuma Ohara[†] Hikaru Tsuchida[†]

October 2, 2018

Abstract

Protocols for secure multiparty computation (MPC) enable a set of mutually distrusting parties to compute an arbitrary function of their inputs while preserving basic security properties like *privacy* and *correctness*. The study of MPC was initiated in the 1980s where it was shown that any function can be securely computed, thus demonstrating the power of this notion. However, these proofs of feasibility were theoretical in nature and it is only recently that MPC protocols started to become efficient enough for use in practice. Today, we have protocols that can carry out large and complex computations in very reasonable time (and can even be very fast, depending on the computation and the setting). Despite this amazing progress, there is still a major obstacle to the adoption and use of MPC due to the huge expertise needed to design a specific MPC execution. In particular, the function to be computed needs to be represented as an appropriate Boolean or arithmetic circuit, and this requires very specific expertise. In order to overcome this, there has been considerable work on compilation of code to (typically) Boolean circuits. One work in this direction takes a different approach, and this is the SPDZ compiler (not to be confused with the SPDZ protocol) that takes high-level Python code and provides an MPC run-time environment for securely executing that code. The SPDZ compiler can deal with arithmetic and non-arithmetic operations and is extremely powerful. However, until now, the SPDZ compiler could only be used for the specific SPDZ family of protocols, making its general applicability and usefulness very limited.

In this paper, we extend the SPDZ compiler so that it can work with general underlying protocols. Our SPDZ extensions were made in mind to enable the use of SPDZ for arbitrary protocols and to make it easy for others to integrate existing and new protocols. We integrated three different types of protocols, an honest-majority protocol for computing arithmetic circuits over a field (for any number of parties), a three-party honest majority protocol for computing arithmetic circuits over the ring of integers $\mathbb{Z}_{2^n}$, and the multiparty BMR protocol for computing Boolean circuits. We show that a single high-level SPDZ-Python program can be executed using all of these underlying protocols (as well as the original SPDZ protocol), thereby making SPDZ a true general run-time MPC environment.

[*]An extended abstract of this work appeared at ACM CCS 2018.

[†]NEC, `t-araki@ek.jp.nec.com, k-ohara@ax.jp.nec.com, h-tsuchida@bk.jp.nec.com`

[‡]Dept. of Computer Science, Bar-Ilan University, Israel. `assaf.barak@biu.ac.il,lindell@biu.ac.il`. Supported by the European Research Council under the ERC consolidators grant agreement n. 615172 (HIPS), by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office, and by the Alter Family Foundation.

[§]NEC Israel Research Center, `jun.furukawa@necam.com`

[¶]Data61, `mks.keller@gmail.com`

In order to be able to handle both arithmetic and non-arithmetic operations, the SPDZ compiler relies on conversions from field elements to bits and back. However, these conversions do not apply to ring elements (in particular, they require element division), and we therefore introduce new bit decomposition and recomposition protocols for the ring over integers with replicated secret sharing. These conversions are of independent interest and utilize the structure of $\mathbb{Z}_{2^n}$ (which is much more amenable to bit decomposition than prime-order fields), and are thus much more efficient than all previous methods.

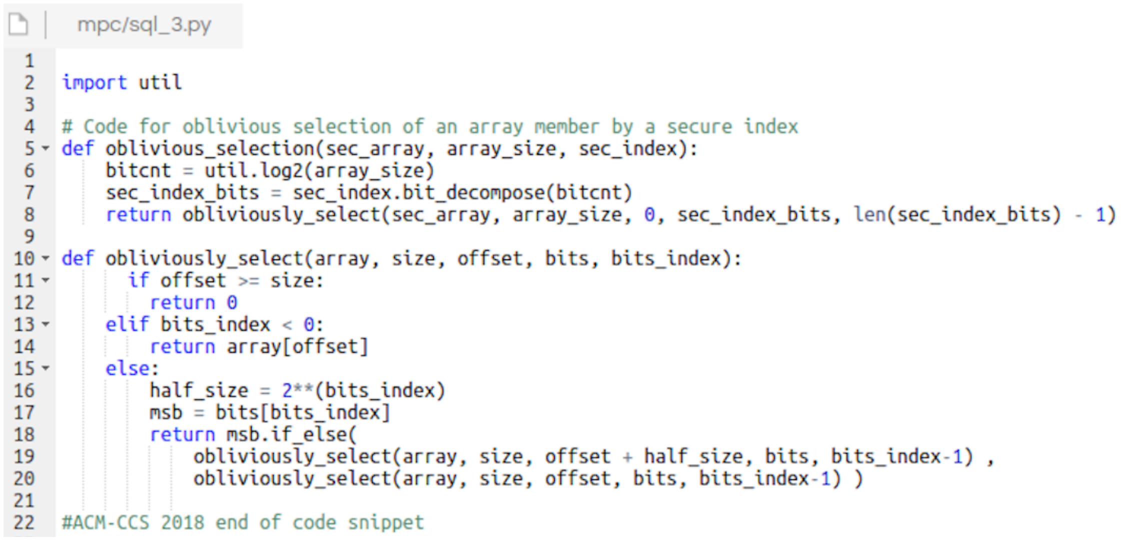
We demonstrate our compiler extensions by running a complex SQL query and a decision tree evaluation over all protocols.

1 Introduction

Background. Secure multiparty computation (MPC) protocols provide the capability of computing over private data without revealing it. In today's privacy crisis, MPC can serve as a core tool for utilizing data without compromising the security and privacy of an individual's sensitive information. In recent years there has been immense progress in the efficiency of MPC protocols, and today we can securely compute large Boolean and arithmetic circuits representing real computations of interest. However, most MPC protocols require the description of a Boolean and/or arithmetic circuit in order to run. This is a significant obstacle in the deployment of MPC, since circuits for real problems of interest can be very large and very hard to construct. In order to deal with this, there has been quite a lot of work on compiling high-level programs to circuits; see [15, 29, 8, 9] for just a few examples. However, many of these works are limited in the size of the circuit that they can generate, and most of them do not deal with the general problem of combined arithmetic and non-arithmetic (Boolean) computations. In addition, the paradigm of working with static circuits is problematic for huge computations, due to the size of the circuit that must be dealt with (this issue has been considered in [29] and elsewhere, but can still be an issue).

The SPDZ protocol and compiler. In contrast to the above, the series of works called "SPDZ" took a very different approach. SPDZ is the name of a specific protocol for honest-minority multiparty computation [14]. However, beyond improvements to the protocol itself, follow-up work on SPDZ included the implementation of an extremely powerful MPC run-time environment/compiler that is integrated into the SPDZ low-level protocol [13, 21, 7]. From here on we differentiate between the *SPDZ protocol* which is a way of executing secure MPC over arithmetic circuits, and the *SPDZ compiler* that is a general run-time environment that takes code written in a high-level Python-type language, and executes it in MPC over the SPDZ protocol. We stress that SPDZ does not generate a circuit and hand it down to the low-level protocol. Rather, it behaves more like an interpreter, dynamically calling the lower-level protocol to carry out low-level operations.

A key property of the SPDZ compiler is that it separates the basic operations provided by MPC protocols (binary or arithmetic circuits) from a protocol (or program) using those operations as building blocks. While the basic operations mostly consist of simple arithmetic over some ring (more precisely, a field in case of SPDZ), combining them to achieve higher-level operations, like integer or fixed-point division, is a more complex matter. However, integrating such higher-level operations into the core MPC engine is not a good strategy because the reduction to basic operations is likely very similar even for different underlying protocols. The SPDZ compiler provides a tool to write more complex building blocks, which then can be used in arbitrary MPC applications without being concerned about the details of those blocks nor the underlying protocol. A concrete example of the ease in which complex secure computations can be specified appears in Figure 1. This program describes the task of selecting an element from an array, where both the array values *and* the array index are private (and thus shared). Given that the size of the array is also a variable, this is very difficult to specify in a circuit. This highlights another huge advantage of this paradigm. The SPDZ system facilitates modular programming techniques, enabling the software engineer to program functions that can be reused in many programs. (Note that a simpler linear program could be written for the same task, but this method is more efficient. Observe the richness of the language, enabling recursion, if-then-else branching, and so on.)


mpc/sql_3.py

```

import util

# Code for oblivious selection of an array member by a secure index
def oblivious_selection(sec_array, array_size, sec_index):
    bitcnt = util.log2(array_size)
    sec_index_bits = sec_index.bit_decompose(bitcnt)
    return obliviously_select(sec_array, array_size, 0, sec_index_bits, len(sec_index_bits) - 1)

def obliviously_select(array, size, offset, bits, bits_index):
      if offset >= size:
        return 0
    elif bits_index < 0:
        return array[offset]
    else:
        half_size = 2**(bits_index)
        msb = bits[bits_index]
        return msb.if_else(
            obliviously_select(array, size, offset + half_size, bits, bits_index-1) ,
            obliviously_select(array, size, offset, bits, bits_index-1) )

#ACM-CCS 2018 end of code snippet
```

Figure 1: SPDZ Python code for oblivious selection from an array.

In Appendix A we demonstrate the ease with which the SPDZ language can be used, by documenting the timeline from the specification of a complex SQL query to implement at 8:45am, to the pseudocode prepared by the researchers at 11:16am, to a working version of the SQL program inside SPDZ at 2:49pm, all on the same day. We stress that this was the first time a program of this type was constructed in our lab, and so this accurately reflects the time to develop (of course, after gaining general experience with the SPDZ compiler).

Extending the SPDZ compiler. Prior to our work, the SPDZ compiler was closely integrated with the SPDZ low-level protocol, preventing its more broad use. The primary aim of this work is to extend the SPDZ compiler so that other protocols can be integrated into the system with ease. This involved making changes to the SPDZ compiler at different levels, as is described in Section 3. In order to demonstrate the strength of this paradigm, we integrated three different protocols of completely different types. Specifically, we integrated the honest-majority multiparty protocol of [24] for arithmetic circuits over a field, the three-party honest-majority protocol of [4, 3] for arithmetic circuits over the ring $\mathbb{Z}_{2^n}$ for any n, and the BMR protocol [6, 25] for constant-round multiparty computation for Boolean circuits. The integration of the former protocol required the fewest number of changes, since it works over any field just like the original SPDZ, whereas the other protocols required more changes. For example, the SPDZ compiler already comes with high-level algorithms for fixed-point and floating point operations, integer division and more. All of these are reusable as-is for any other protocol based on fields. However, for protocols over the ring $\mathbb{Z}_{2^n}$, different high-level algorithms needed to be developed. We have done this, and thus other protocols over rings can utilize the relevant high-level algorithms.

We stress that the focus of our extensions were not to integrate these specific protocols, but to modify the SPDZ system in order to facilitate easy integration of other protocols by others. We believe that this is a significant contribution, and will constitute a step forward to enabling the widespread use of MPC.

Bit decomposition and recomposition. The advantage of working over arithmetic circuits (in contrast to Boolean circuits) is striking for computations that require a lot of arithmetic, as is typical for computing statistics. In these cases, addition is for free, and multiplication of large values comes at a cost of a single operation. However, most real-world programs consist of a combination of arithmetic and non-arithmetic computations, and thus need a mix of arithmetic and Boolean low-level operations. In order to facilitate this, it is necessary to have *bit decomposition* and *recomposition* operations, to convert a shared field/ring element to a series of shares of its bit representation and back. This facilitates all types of computation, by moving between the field/ring representation and bit representation, depending on the computation. For example, consider an SQL query which outputs the average age of homeowners with debt above the national average, separately for each state. This requires computing the national debt average (arithmetic), comparing the debt of each homeowner with the national average (Boolean) and computing the average age of those whose debt is greater (mostly arithmetic for computing the sum, and one division for obtaining the average). Note that the last average requires division since the number of homeowner above the average is not something revealed by the output, and division is computed using the Goldschmidt method which requires a mix of arithmetic and bit operations, including conversions.

As we discuss below in Section 2, the SPDZ compiler includes high-level algorithms for many complex operations, and as such includes bit decomposition and recomposition. In some cases, the operations rely on division in the field and so cannot be extended to rings. In order to facilitate working with rings, we therefore develop novel protocols for bit decomposition and ring recomposition between $\mathbb{Z}_{2^n}$ and $\mathbb{Z}_2$ that are based on replicated secret sharing and therefore compatible with [4, 3]. Since $\mathbb{Z}_{2^n}$ preserves the structure of the individual bits much more than $\mathbb{Z}_p$ for a prime p, it is possible to achieve *much faster* decomposition and recomposition than in the field case. Thus, in programs that require a lot of conversions, ring-based protocols can way outperform field-based protocols. However, field-based protocols are typically more efficient for the basic arithmetic (e.g., compare the ring version of [3] to [24]). Thus, different low-level protocols have different performance for different programs. Stated differently, there is no "best" protocol, even considering a specific number of parties and security level, since it also depends on the actual operations carried out (this is also true regarding deep vs shallow circuits, and constant versus non-constant round protocols). This gives further justification to have a unified SPDZ system that can work with many low-level protocols *of different types*, so that a program can be written once and tested over different protocols in order to choose the best one.

Our protocols for bit decomposition and ring recomposition are described in Section 4.

Implementation and experiments. In Section 5, we present the results of experiments we ran on programs for evaluating unbalanced decision trees (this is more complex than balanced decision trees due to the need for the evaluation to be completely oblivious) and for evaluating complex SQL queries. Although the focus of our work is not efficiency, we report on running times and comparisons in order to provide support for the fact that this SPDZ extension is indeed very useful and meaningful.

Our implementation is open-source and available for anyone interested in utilizing it.

2 The SPDZ Protocol and Compiler

2.1 Overview

SPDZ is the name given to a multiparty secure computation protocol by Damgård et al. [14, 13] that works for any n parties. It provides active (malicious) security against any $t \leq n$ corrupted parties, and it works in the preprocessing model, that is, the computation is split into a data-independent ("offline") and a data-dependent ("online") phase. The main idea of SPDZ is to use relatively expensive somewhat homomorphic encryption in the offline phase while the online phase purely relies on cheaper modular arithmetic primitives. This also allows for an optimistic approach to the distributed decryption used in the offline phase: Instead of proving correct behavior using zero-knowledge proofs, the parties check the decrypted value for correctness and abort in case of an error. Nevertheless, there is no leakage of secret data because no secret data has yet been used.

The main link between the two phases is a technique due to Beaver [5], which reduces the multiplication of secret values to a linear operation on secret values using a precomputed multiplication of random values and revealing of masked secret-shared values. Using a linear secret sharing scheme makes this technique straightforward to use. Additive secret sharing is trivially linear, and it provides the desired security against any number of $t \leq n$ corrupted parties. On the top of additive secret sharing, SPDZ also uses an information-theoretic tag (the product of the secret value and a global secret value), which is additively secret-shared as well, thus preserving the linear property.

Keller et al. [21] have created software to run the online phase of any computation, optimizing the number of communication rounds. The software receives a description of the computation in a high-level Python-like language, which is then compiled into a concise byte-code that is executed by the SPDZ virtual machine (which includes the actual SPDZ MPC protocol); see Figure 2. The design of the virtual machine follows the design principles of processors by providing instructions such as arithmetic over secret-shared or public values (and a mix between them), and branching on public values. The inclusion of branching means that one can implement concepts common in programming languages such as loops, if-else statements, and functions. While the conditions for loop and if statements can only depend on public values,[1] this provides an obvious benefit in reducing the representation of a computation and the cost of the optimization described below. In particular, it is possible to loop over a large set of inputs without representing the whole circuit in memory. We call this software layer the ***SPDZ compiler***, in order to distinguish it from the ***SPDZ protocol***. We remark that although the SPDZ compiler was developed with the SPDZ protocol specifically in mind, its good design enabled us to extend it to other protocols and make it a general MPC tool.

[1]This is an inherent requirement for "plain" multiparty computation. There are solutions that overcome this [18], but they come with a considerable overhead.

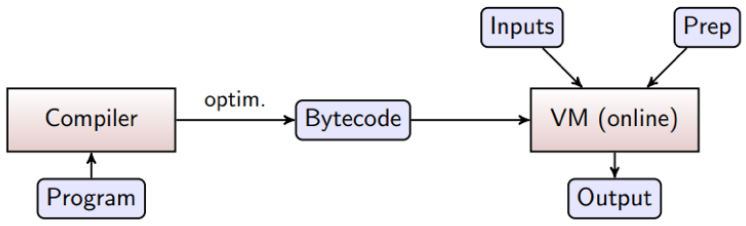


Figure 2: High-level SPDZ compiler architecture.

2.2 Circuit Optimizations

The core optimization of the software makes use of the fact that, using Beaver's technique, the only operation that involves communication is the revealing of secret values. This means that the compiler can merge all operations in a single communication round into a single opening operation, effectively reducing the communication to the minimum number rounds for a given circuit description. In addition, the software splits the communication into two instructions to mark the sending and the receiving of information, and optimizes by placing independent computations in-between the send-and-receive, providing the ability to use the time that is required to wait for information from the other parties. As such, all opening operations are framed between `start` and `stop` instructions, and independent instructions that can be processed in parallel to the communication are placed between them. For example, `startopen` denotes the beginning of a series of instructions to open (reveal) shares, and `stopopen` denotes the end of the series.

In order to achieve the above effect of grouping all communication messages per round, the SPDZ compiler represents the computation as a directed acyclic graph where every instruction is represented as a node, and nodes are connected if one instruction uses another's output as input. The vertices are assigned weight *one* if the source instructions start a communication operation and *zero* otherwise. The communication round of any instruction is then the longest path from any source with respect to the vertex weights. It is straight-forward to compute this by traversing the instructions in order and assigning the maximum value of all input vertices to each instruction.

It is important to note that merging all instructions that can be run in parallel needs to be done carefully. In particular, it does not suffice to merge the open instructions that are independent of each other, but also any operations that the open depends on. This is solved by computing the topological order of the changed graph, and by adding vertices between instructions with side effects, in order to maintain the order between them. This results in a trade-off because adding more vertices in order to preserve the order can lead to more communication rounds.

The optimization described in this subsection (of reducing the number of communication rounds) is only possible on a straight-line computation without any branching. We therefore perform this optimization separately on each part of the computation of maximal size. These components are called basic blocks in the compiler literature.

2.3 Higher-Level Algorithms

In terms of arithmetic operations, the virtual machine provides algebraic computations on secret values provided by the MPC protocol (addition and multiplication) and general field arithmetic on public values (such as addition, subtraction, multiplication, and division). This clearly does not suffice for a general, easy to use programming interface. Therefore, in addition to the Beaver's

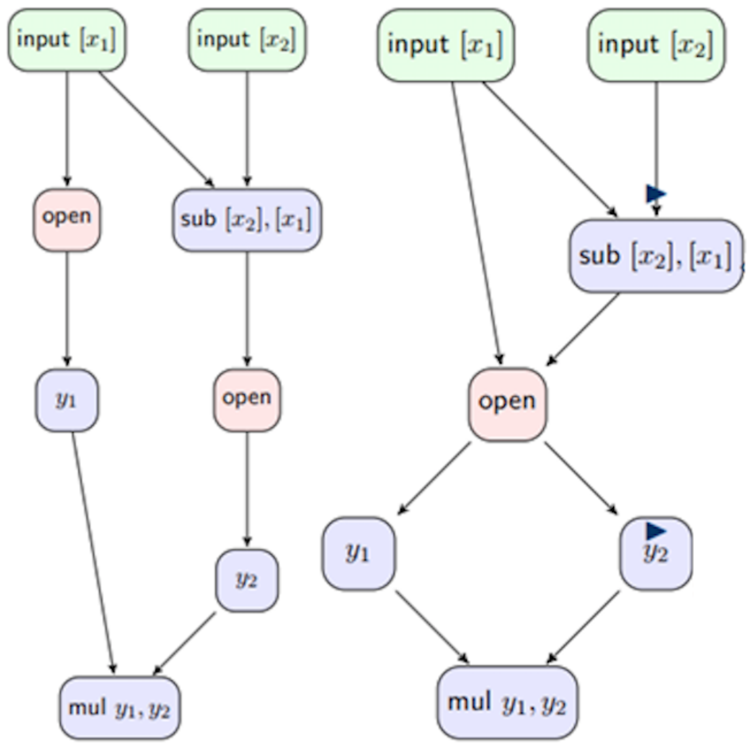


Figure 3: Representing a program as a directed acyclic graph.

technique for secret-value multiplication described above, the compiler comes with a library that provides non-algebraic operations on secret values such as *comparison* (equals, less-than, etc.), and arithmetic for both floating- and fixed-point numbers. This library is based on a body of literature [11, 10, 2] that uses techniques such as statistical masking to implement such operations without having to rely solely on field-arithmetic circuits.[2]

The following bit decomposition of a secret-sharing of $0 \le x < 2^m$ for some m illustrates the nature of these protocols. Assume that $[x]$ is a secret-sharing of a x in a field $\mathbb{F}$ such that $2^{m+k} < |\mathbb{F}|$, with k being the statistical security parameter. Let r be a random value such that $0 \le r < 2^{m+k}$, consisting of bits r_i for $i = 0, \ldots, m+k-1$, and let $[r_0], ..., [r_{m+k-1}], [r]$ be their secret sharings, all over the field $\mathbb{F}$. (It is possible to generate these shares by sampling $[r_0], ..., [r_{m+k-1}]$ in the offline phase, as discussed in [13], and then computing $[r] = \sum [r_i] \cdot 2^i$ locally.) Similarly, we can compute $[z] = [x + r]$ from $[r]$ and $[x]$ locally. Observe that z statistically hides x because the statistical distance between the distributions of z and of r is negligible in k. Therefore, we can reveal z and decompose it into bits $z_0, ..., z_{m+k-1}$. Finally, the shares of the bits of x, $([x_0], ..., [x_{m-1}])$, can be computed from $(z_0, ..., z_{m+k-1})$ and $([r_0], ..., [r_{m+k-1}])$ via a secure computation of a Boolean circuit.

The compiler also provides the same arithmetic interface when using the SPDZ protocol with a finite field of characteristic two, allowing the execution of the same computation on different underlying protocols. We used this as a stepping stone for the extension using garbled circuits

[2]Arithmetic circuits are essentially polynomials, and a naive implementation of an operation like the comparison of numbers in a large field is very expensive.

below because of the similarity between them.

Implementing these algorithms at this level rather than within the virtual machine below has the advantage that all optimizations in the compiler are automatically applied to any MPC VM. We stress that these algorithms are part of the SPDZ compiler layer.

3 Making SPDZ a General Compiler

In order to generalize the SPDZ compiler to work for other protocols, modifications needed to be made at multiple levels. Our aim when designing these changes was to make them as general as possible, so that other protocols can also utilize them. We incorporated three very different protocols in order to demonstrate the generality of the result:

1. *Honest-majority MPC over fields:* We incorporated the recent protocol of [24] that computes arithmetic circuits over any finite field, assuming an honest majority. This protocol has a direct multiplication operation, and does not work via triples like the SPDZ protocol. (The protocol does use triples in order to prevent cheating, but not in a separate offline manner.) The specific protocol incorporated works over $\mathbb{Z}_p$ with Mersenne primes $p = 2^{61} - 1$ or $p = 2^{127} - 1$.

2. *Honest-majority MPC over rings:* We incorporated the three-party protocol of [4, 3] that computes arithmetic circuits over any *ring* including the ring $\mathbb{Z}_n$ of integers for any $n \geq 1$. The fact that this protocol operates over a ring and not a field means that it is not possible to divide values; this requires changing the way many operations are treated, as will be described below.

3. *Honest minority MPC for Boolean circuits:* We incorporated a protocol for computing any Boolean circuit using the BMR paradigm [6]. Our starting point for this purpose was the software of [23] that was developed for a different purpose; we therefore made the modifications needed for our purpose.

We discuss these different protocols in more detail below. In this section, we describe the changes that we made to the SPDZ compiler in order to enable other protocols to be incorporated in it, with specific examples from the above.

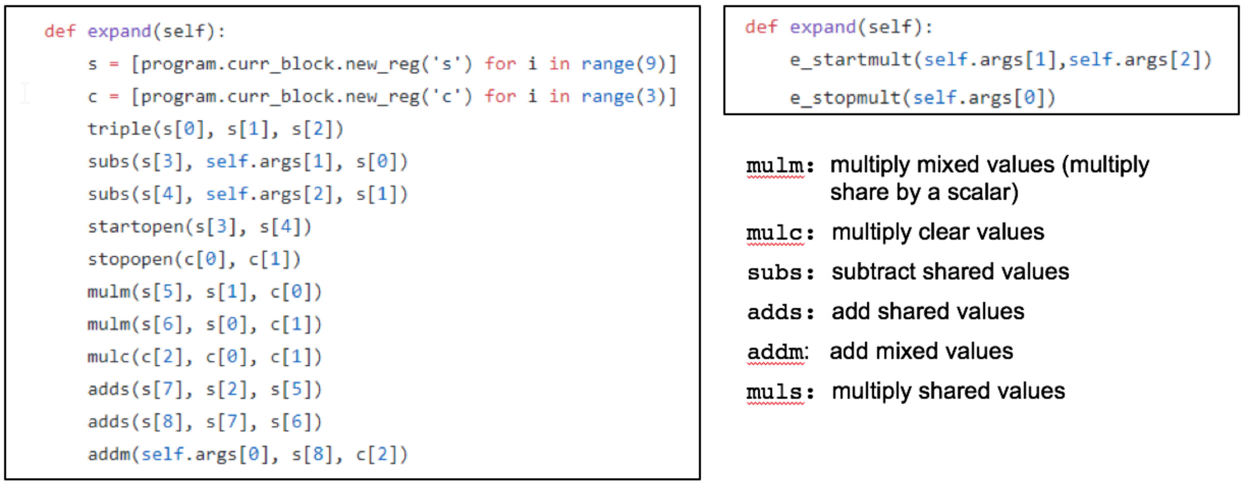
3.1 Modifications to the SPDZ Compiler

Infrastructure modifications at the compiler level. The main difficulty in adapting the SPDZ compiler to other protocols lies in the fact that many protocols do not use the Beaver technique, and so do not reduce secret value multiplications to the opening of masked values only. Such protocols include secret-value multiplications as an atomic operation of the protocol, and thus working via `startopen` and `stopopen` only would significantly reduce the protocol's performance.[3] We therefore generalized the communication pattern of the compiler to allow general communication and arbitrary pairs of start/stop instructions for communication, rather than specifically supporting only start/stop of share openings. Our specific protocols have atomic multiplication operations that involve communication, and so we specifically added `e_startmult` and `e_stopmult` (which are start and stop of multiplication operations, where the `e`-prefix denotes an extension), but our generalization allows adding any other type of communication as well.

[3] We stress that the only communication in the SPDZ protocol is in the opening of shared values, and all other operations – including multiplication – are reduced to local computation and opening.

Since the multiplication within the protocols that we added involves communication, it is desirable to merge as many multiplications as possible with the reveal (or open) operations in the SPDZ compiler. The fork of the SPDZ compiler used by Keller and Yanay for their BMR implementation [22, 23] provides functionality to merge several kinds of instructions separately (AND and XOR in their case). We used this for multiplication and open instructions, resulting in circuit descriptions that minimize the number of multiplication and open rounds separately. This is not optimal because it does not provide full parallelization of the communication incurred by multiplication and open operations that could be carried out in parallel. However, we argue that this is sufficient because in protocols that support atomic multiplication, opening is typically only necessary at the end of a computation that involves many rounds of multiplications.

Algorithm modifications at the compiler level. The modular construction of the compiler and the algorithm library allows us to re-use many higher-level protocols mentioned in the previous section. These algorithms are represented in the compiler as expansions of an operation. For example, multiplication of shared values in the SPDZ protocol is a procedure that utilizes a multiplication triple, and carries out a series of additions, subtractions and openings in order to obtain the shared product. This algorithm is replaced by a single `startmult` and `stopmult` using our new instructions for low-level MPC protocols that have an atomic multiplication operation. See Figure 4 for code comparison.



```
def expand(self):
    s = [program.curr_block.new_reg('s') for i in range(9)]
    c = [program.curr_block.new_reg('c') for i in range(3)]
    triple(s[0], s[1], s[2])
    subs(s[3], self.args[1], s[0])
    subs(s[4], self.args[2], s[1])
    startopen(s[3], s[4])
    stopopen(c[0], c[1])
    mulm(s[5], s[1], c[0])
    mulm(s[6], s[0], c[1])
    mulc(c[2], c[0], c[1])
    adds(s[7], s[2], s[5])
    adds(s[8], s[7], s[6])
    addm(self.args[0], s[8], c[2])
```

```
def expand(self):
    e_startmult(self.args[1],self.args[2])
    e_stopmult(self.args[0])
```

mulm:	multiply mixed values (multiply share by a scalar)
mulc:	multiply clear values
subs:	subtract shared values
adds:	add shared values
addm:	add mixed values
muls:	multiply shared values

Figure 4: Multiplication in the original SPDZ compiler vs using new instruction extension.

For the case of our honest-majority field-based protocol, this is the only change that we needed to make to the compiler. This is because all of the original SPDZ compiler algorithms (e.g., for floating and fixed-point operations, integer division, bit decomposition, etc.) work for any field-based MPC, and thus also here. However, when field division is not available, as in the example of the ring-based protocol, different high-level algorithms needed to be provided. A very important example of this relates to bit decomposition and recomposition for ring-based protocols, which is an operation needed for many higher-level arithmetic operations including non-algebraic operations like comparison. We present a new highly-efficient method for bit decomposition and recomposition over $\mathbb{Z}_{2^n}$ in Section 4, and this was incorporated on the algorithm level. In addition, new algorithms were added for fixed-point multiplication and division, integer division, comparison, equals, and more.

We stress that once the infrastructure modifications were made, all of these changes are algorithmic only, meaning that they rewrite the `expand` operation that converts a high-level algorithm into a series of low-level supported operations (like multiplication in Figure 4).

Modifications to bytecode. The bytecode that is generated by the compiler includes the low-level instructions and opcodes supported by the MPC protocol itself. As such, some changes were needed to add new instructions and opcodes supported by the other MPC protocols. Thus, a direct multiplication opcode needed to be added (for both the field and ring protocols), as well as some additional commands for the bit decomposition and recomposition needed for the ring protocol (e.g., the local decomposition steps described in Sections 4.3 and 4.4). Finally, a new `verify` command was added since the honest-majority field and ring protocols do not use SPDZ MACs and verify correctness in a different way. The bytecode also includes a lot of instructions needed for jumping, branching, merging threads and so on. Fortunately, all of this can be reused as is, without any changes.

Modifications to the virtual machine. On the level of the virtual machine, we have modified the SPDZ compiler software to call the relevant function of an external library for every instruction that involves secret-shared values. This comes down to roughly twenty instructions. We have done this in a way that facilitates plugging in other backend libraries, which allows us to easily run the same program using different protocols. This is in line with our goal of enabling the same high-level interface to be used to program for completely different MPC schemes. For the field case, the changes here were relatively small. The multiplication was changed, but so was scalar addition since in the SPDZ protocol each party carries out the same operation locally, whereas different parties act differently for scalar addition in the replicated secret-sharing protocol version of [24]. In addition, triple generation is not carried out offline but done on demand, and the MAC was disabled at the VM level (i.e., an extension was added to optionally disable the MAC so that the VM is compatible both with protocols that use and do not use MACs). Finally, the original SPDZ protocol relies on Montgomery multiplication [26]. While this is efficient for general moduli, in some cases like when using Mersenne primes, more efficient modular multiplication can be achieved directly. The field protocol implementation of [24] utilizes Mersenne primes, and this was therefore also integrated into the VM interface.

For the ring protocol, there were more changes required since the division of clear elements is not supported in a ring, and since more instructions are needed at the basic protocol level (for decomposition and recomposition, as described above). In addition, the `input` procedure was changed since the secret sharing is different. We stress that these are not just at the protocol level since the VM uses special registers for local operations (to improve performance) and so these need to be modified.

We remark that the compiler, bytecode, and VM needed to be very significantly modified for the Boolean circuit (BMR) protocol, and thus a separate branch was created. This is understandable since the protocol is of a completely different nature. Nevertheless, the key property that it *all runs under the same MPC program high-level language* is achieved, and thus to the “MPC user” writing MPC programs, this is not noticeable.

Explanation of Figure 5. In Figure 5 we present a diagram illustrating the different extensions to the SPDZ compiler, for all three protocols incorporated. On the left, the original SPDZ architecture is presented. Then, the field-based protocol of [24] is presented, with relatively minor changes (mainly adding the multiplication extension); of course, the MPC protocol at the lower level is replaced as well. Next, the ring-based protocol is presented, and it includes more modifications, including support for different types of shares and numbers, as well as more modifications to the compiler and below. There are also some additions to the language itself, since adding explicit instructions like `inject` (which maps a bit into a ring element) improves the quality of the compiler. Finally, the architecture for the BMR protocol is added; as stated above, this requires major changes throughout, except for the programmer interface and language which remain the same.

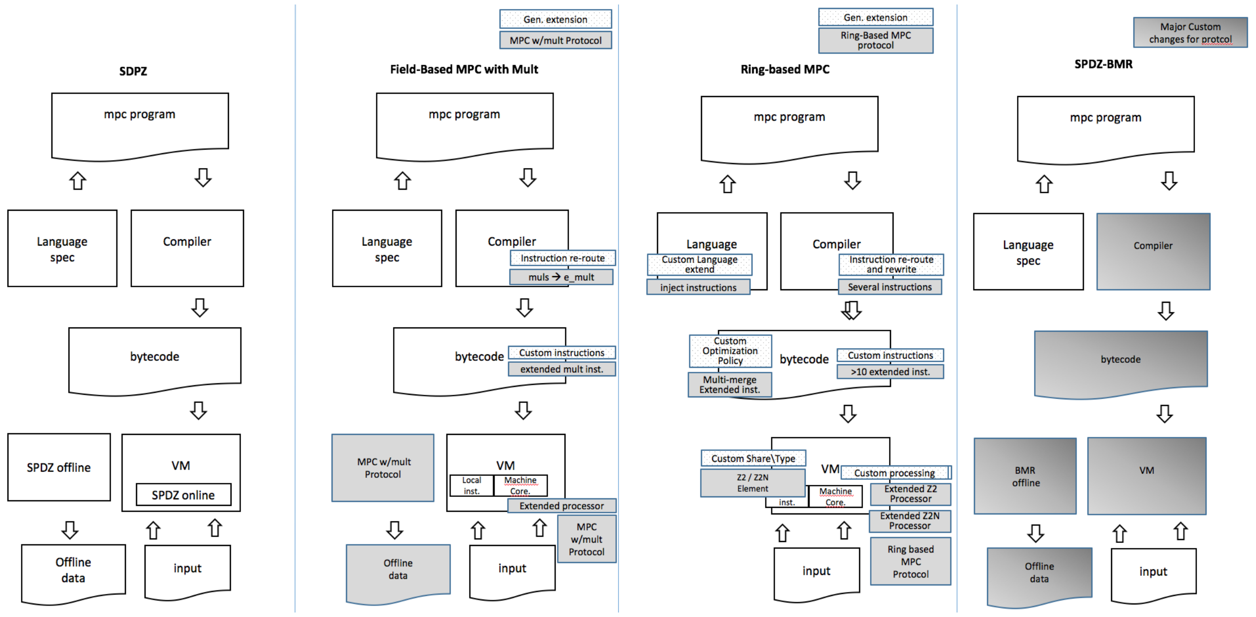


Figure 5: The extensions applied to the SPDZ compiler of [13].

3.2 Incorporating BMR Circuits

In this section, we provide additional details about the incorporation of the BMR Boolean circuit protocol into SPDZ. Since the original SPDZ protocol is based on *secret sharing* and *arithmetic circuits*, the changes required to incorporate a garbled-circuit based protocol were the most significant.

In order to evaluate our programs in a garbled circuit setting, we have made use of the recently published software implementing oblivious RAM [22, 23] in the SPDZ-BMR protocol [25]. The latter denotes the combination of BMR, which is a method of generating a garbled circuit using any MPC scheme, with the SPDZ protocol [14] as the concrete scheme. While there are recent protocols achieving similar goals [31, 17], we would argue that the BMR software is the most powerful one publicly available to date, and that it still gives a reasonable indication of the performance of garbled circuits with active security.

The software follows the same paradigm as SPDZ in that it implements a virtual machine that executes bytecode consisting of instructions for arithmetic, branching, input/output, etc. The main difference is that arithmetic here means XOR and AND. Furthermore, while the smallest units at the virtual machine level are secret-shared and public values in a field for SPDZ, here they are vectors of secret-shared bit and public values. This leads to more concise circuit descriptions. Furthermore, the compiler merges several types of instructions to further vectorize instructions, which may reduce the number of communications rounds (e.g., for inputs), enable the use of several processor cores, and facilitate pipelining of AES-NI instructions when evaluating as many AND gates in parallel as possible.

The primary goal of the software of [22, 23] is the evaluation of ORAM. Hence we needed to extend it in various aspects, most notably the following:

Private inputs: This feature was omitted from [22, 23] who wished only to evaluate the performance of computation. In our context however, private inputs play a major role. This change mostly affected the virtual machine of the BMR implementation.

Arithmetic: While the software of [22, 23] contains provisions for integer arithmetic in fields of characteristic two (and thus for binary circuits) and for fixed-point calculations in arithmetic circuits, we had to combine and supplement this for our purposes. In particular, it turned out that the translation of fixed-point division from arithmetic to binary circuits is non-trivial because keeping exact track of bit lengths is vital in the latter. Since the virtual machine only deals with binary circuits by design, this change was exclusively on the compiler side.

The software is incomplete in the sense that it only implements the evaluation phase securely, while the use of the SPDZ protocol in the garbling part is simulated using a separate program. Nevertheless, the evaluation timings are accurate because the garbled circuit is read from solid-state disks. Furthermore, the uniform nature of the circuit generation as well as the offline phase of SPDZ (called function-dependent phase in this context) allowed us to micro-benchmark the two phases. For the latter, this has been done in various previous works [20, 13].

4 Protocols for Rings with Replicated Secret Sharing

As we have discussed above, the SPDZ compiler provides high-level algorithms for operations from numerical comparisons to fixed and floating point computations. These algorithms require the capability to decompose a basic element into its bit representation and back. Since the SPDZ protocol works over fields, it already contains these methods for field elements. However, it does *not* support bit decomposition and recomposition for ring elements. Since this is crucial for running SPDZ programs over ring-based MPC, in this section we describe a new method for bit decomposition and recomposition for the ring-based protocol of [4, 3]. We stress that our method works for any 3-party protocol based on replicated secret sharing as is the case for [4, 3], but it does not work for any ring-based protocol in general. We follow this strategy in order to achieve highly efficient bit decomposition and recomposition; since these operations are crucial and ubiquitous in advanced computations, making the operation as efficient as possible is extremely important.

Before beginning, we explain why bit decomposition and recomposition can be made much more efficient in the ring $\mathbb{Z}_{2^n}$. Consider the case of additive shares where the parties hold values s_i such that $\sum_{i=1}^{n} s_i = s$, where s is the secret. If the addition is in a field like $\mathbb{Z}_p$, then the values of all bits depend on all other bits. In particular, the value of the least significant bit depends also on the most significant bits; consider computing $16 + 8 \bmod 17$. The three least significant bits of 16 and 8 are zero, but the result is 7, which is 111 in binary. This is not the case in $GF[2^n]$ and bit decomposition is actually trivial in this field. However, since we typically use arithmetic circuits to embed numerical computations, we need integer addition and multiplication to be preserved in the field or ring. For this reason, the ring $\mathbb{Z}_{2^n}$ has many advantages. First, it allows for very efficient local operations. Second, the sum of additive shares has the property that each bit of the result depends only on the corresponding bit in each share, and the carry from the previous share. We will use this in an inherent way in order to obtain more efficient bit decomposition and recomposition protocols.

4.1 The Challenge and Our Approach

Efficient bit decomposition and ring composition are essential primitives for efficient MPC, since many real-world programs require both arithmetic computations, as well as comparison and other operations that require bit representation. However, such conversions are difficult to carry out, especially in the presence of malicious adversaries. This is due to the fact that malicious parties can change the values that they hold, and a secure protocol has to prevent such behavior. We overcome this by constructing protocols that are comprised of *only* standard ring-MPC operations (over shares of ring elements), standard bit-MPC operations (over shares of bits), and local transformations from *valid* ring-shares to *valid* bit-shares (and vice versa) that are carried out independently by each party. Since this is the case, the security is easily reduced to the security of the ring and bit protocols which have been proven.

Our bit-decomposition and ring-composition conversion protocols are constructed specifically for replicated secret sharing and between the ring $\mathbb{Z}_{2^n}$ (for any n) and $\mathbb{Z}_2$. Although this is a very specific scenario, it enables very high throughput secure computation of any functionality (in the setting of three parties, with at most one corrupted). In particular, the recent protocols of [4] and [3] can be used. These protocols achieve high throughput by requiring very low communication: in the protocol of [4] for semi-honest adversaries, each party sends a single bit (resp., ring element) per AND gate (resp., multiplication gate) when computing an arbitrary Boolean circuit (resp., arithmetic circuit over $\mathbb{Z}_{2^n}$). Furthermore, the protocol of [3] achieves security in the presence of malicious adversaries in this setting at the cost of just 7 times that of [4] (i.e., 7 bits/ring elements per AND/multiplication gate).

Our method utilizes local computations and native multiplications and additions in Boolean and ring protocols. As such, if the underlying Boolean and ring protocols are secure for *malicious adversaries*, then the result is bit decomposition and recomposition that is secure for *malicious adversaries*. Likewise, if the underlying protocols are secure for semi-honest adversaries then so is the result.

4.2 Replicated Secret Sharing

Let P_1, P_2, P_3 be the participating parties. We consider the ring $\mathbb{Z}_{2^n}$ of n-bit integer operations (modulo 2^n). A highly efficient 3-party protocol for working over $\mathbb{Z}_{2^n}$ with security in the presence of semi-honest adversaries was presented in [4]. The protocol of [4] for the Boolean case was extended to deal with malicious adversaries in [16, 3]. However, the exact same methods work to achieve security for malicious adversaries for the case of $\mathbb{Z}_{2^n}$ as well. These protocols use replicated secret sharing, defined as follows.

Let $x \in \mathbb{Z}_{2^n}$ be an element. In order to generate a three-party sharing of x, first choose random $x_1, x_2 \in \mathbb{Z}_{2^n}$ and define $x_3 = x - x_1 - x_2 \bmod 2^n$ (all operations are modulo 2^n and we will therefore omit this from hereon). Observe that $x = x_1 + x_2 + x_3$ and thus these are *additive shares*. The replicated secret sharing is such that each P_i receives the pair (x_i, x_{i-1}). (We will write $i-1$ and $i+1$ in a free manner, and the intention is that the value wraps around between $1, 2, 3$.) We denote a replicated secret sharing of $x \in \mathbb{Z}_{2^n}$ by $[x]_n$. In [4, Section 2.3], it is shown that addition gates can be computed locally, and multiplication gates can be computed semi-honestly with each party sending just a single ring element to one other party. Using the methods of [16, 3], multiplication gates can be computed with malicious security with each party sending just 7 ring elements to one other party.

4.3 Bit Decomposition

Ring operations are extremely efficient for computing sums and products. However, in many cases, it is necessary to also carry out other operations, like comparison, floating point, and so on. In such cases, it is necessary to first *convert* the shares in the ring to shares of *bits*. For example, we can efficiently compute comparison (e.g., less-than) using a Boolean circuit, but we first need to hold the value in Boolean representation. This operation is called bit decomposition. Recall that a sharing of $x \in \mathbb{Z}_{2^n}$ is denoted by $[x]_n$ (and thus a sharing of a bit a is denoted by $[a]_1$). Writing $x = x^n \cdots x^1$ (as its bitwise representation with x^1 being the least significant bit), the bit decomposition operation is a protocol for converting a sharing $[x]_n$ of a single *ring element* $x \in \mathbb{Z}_{2^n}$ into n shares $[x^1]_1, \ldots, [x^n]_1$ of its bit representation. We stress that it is not possible for each party to just locally decompose its shares into bits, because the addition of single bits results in a carry. To be concrete, assume that $x = 1101_2 = 13_{10} \in \mathbb{Z}_{2^4}$ (where subscript of 2 denotes binary, and a subscript of 10 denotes decimal representation). Then, an additive sharing of x could be $x_1 = 1011_2 = 11_{10}$, $x_2 = 1001_2 = 9_{10}$ and $x_3 = 1001_2 = 9_{10}$. If we look separately at each bit of x_1, x_2, x_3, then we would obtain a sharing of $1011_2 = 11_{10} \neq x$ (this is computed by taking the XOR x_1, x_2, x_3).

Step 1 – local decomposition: In this step, the parties locally compute shares of the individual bits of their shares. Let the sharing $[x]_n$ be with values (x_1, x_3), (x_2, x_1) and (x_3, x_2). The parties begin by generating *shares of their shares* x_1, x_2, x_3. This is a local operation defined by the following table:

	P_1	P_2	P_3
Original shares of x:	(x_1, x_3)	(x_2, x_1)	(x_3, x_2)
New sharing of x_1:	$(x_1, 0)$	$(0, x_1)$	$(0, 0)$
New sharing of x_2:	$(0, 0)$	$(x_2, 0)$	$(0, x_2)$
New sharing of x_3:	$(0, x_3)$	$(0, 0)$	$(x_3, 0)$

Observe that each party can locally compute its sharing of the shares, without any interaction. In addition, each sharing is correct. The above local decomposition is actually carried out separately for *each bit* of the shares. Denote by x_i^j the jth bit of x_i where 1 represents the least-significant bit; i.e., $x_i = (x_i^n, x_i^{n-1}, \ldots, x_i^1) \in (\mathbb{Z}_2)^n$. Then, the jth bit of share x_1 is locally converted into the sharing $(x_1^j, 0), (0, x_1^j), (0, 0)$, the jth bit of share x_2 is locally converted into the sharing $(0, 0), (x_2^j, 0), (0, x_2^j)$, and the jth bit of share x_3 is locally converted into the sharing $(0, x_3^j), (0,0), (x_3^j, 0)$. Observe that these are already shares of bits, and are thus actually $[x_1^j]_1, [x_2^j]_1, [x_3^j]_1$. At this point, the parties all hold shares of the bit representation of the shares. This is *not* a bitwise sharing of x, but just of x_1, x_2, x_3. In order to convert these to a bitwise sharing of x, we need to add the shares. However, this addition must take into account the *carry*, and thus local share addition will not suffice.

Step 2 – add with carry: Our aim is to compute the bit representation of $x = x_1 + x_2 + x_3$ using the bitwise shares . Since this addition is modulo 2^n, we need to compute the carry. In the least significant bit, the required bit is just $[x^1]_1 = [x_1^1]_1 + [x_2^1]_1 + [x_3^1]_1 \bmod 2$ (i.e., using local addition of shares). However, we also need to compute the carry, which involves checking if there are at least two ones. This can be computed via the function $\mathsf{majority}(a,b,c) = a \cdot b \oplus b \cdot c \oplus c \cdot a$ which requires 3 multiplications. Since this needs to be computed many times during the bit decomposition, it is important to reduce the number of multiplications. Fortunately, it is possible to compute majority with just a *single* multiplication by

$$\mathsf{majority}(a,b,c) = (a \oplus c \oplus 1) \cdot (b \oplus c) \oplus b.$$

In order to see that this is correct, observe that

$$\begin{aligned}(a \oplus c \oplus 1) \cdot (b \oplus c) \oplus b &= a \cdot (b \oplus c) \oplus c \cdot (b \oplus c) \oplus (b \oplus c) \oplus b \\ &= a \cdot b \oplus a \cdot c \oplus b \cdot c \oplus c \cdot c \oplus b \oplus c \oplus b = a \cdot b \oplus a \cdot c \oplus b \cdot c.\end{aligned}$$

Having computed the carry, it is now possible to compute the next bit, which is the sum of $[x_1^2]_1, [x_2^2]_1, [x_3^2]_1$ *and* the carry from the previous bit. However, observe that since there are now *four* bits to be added, the carry can actually be *two bits*. This in turn means that *five* bits actually need to be added in order to compute the actual bit and to compute its two carries. Denote by c_j and d_j the carries computed from the jth bit. Then, we claim that the bit and its carries can be computed as follows. Compute $[\alpha_j]_1 = [x_1^j]_1 \oplus [x_2^j]_1 \oplus [x_3^j]_1$, $[\beta_j]_1 = \mathsf{majority}\left(x_1^j, x_2^j, x_3^j\right)$ and $[\gamma_j]_1 = \mathsf{majority}\left(\alpha_j, c_{j-1}, d_{j-2}\right)$. Then, compute

$$\begin{aligned}[x^j]_1 &= [\alpha_j]_1 \oplus [c_{j-1}]_1 \oplus [d_{j-2}]_1, \\ [c_j]_1 &= [\beta_j]_1 \oplus [\gamma_j]_1, \quad \text{and} \quad [d_j]_1 = [\beta_j]_1 \cdot [\gamma_j]_1.\end{aligned}$$

(Note that we initialize $c_0 = d_0 = d_{-1} = 0$ for computing x^1, x^2.) In order to see why this computation is correct, observe the following:

1. It is clear that x^j is correct as it is the sum (modulo 2) of the three bits in the jth place, plus the two relevant carry bits from previous places (specifically, c_{j-1} and d_{j-2}).
2. The two carry bits are defined to be $(d_j, c_j) = (\beta_j \cdot \gamma_j, \beta_j \oplus \gamma_j)$. These may equal 00, 01 or 10 in binary (there cannot be a carry of 11 since the maximum sum of 5 bits is 5 which is 101 in binary, resulting in the carry 11).

This is best understood by looking at the table below. We write the result in the last three columns in the order of d_j, c_j, x^j since this is actually the three-bit binary representation of the sum of 5 bits. Since the computation is symmetric between the values of x_1^j, x_2^j, x_3^j and between c_{j-1}, d_{j-2} (meaning that it only matters how many ones there are, but nothing else), it suffices to look only at the number of ones for the x values and the number of ones for c, d.

x_1^j	x_2^j	x_3^j	c_{j-1}	d_{j-2}	α_j	β_j	γ_j	d_j	c_j	x^j
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1
1	1	0	0	0	0	1	0	0	1	0
1	1	1	0	0	1	1	0	0	1	1
0	0	0	1	0	0	0	0	0	0	1
1	0	0	1	0	1	0	1	0	1	0
1	1	0	1	0	0	1	0	0	1	1
1	1	1	1	0	1	1	1	1	0	0
0	0	0	1	1	0	0	1	0	1	0
1	0	0	1	1	1	0	1	0	1	1
1	1	0	1	1	0	1	1	1	0	0
1	1	1	1	1	1	1	1	1	0	1

Observe that the last three columns equal the binary count of the number of ones in the first 5 columns (from 0 to 5), as required. Since the cost of computing majority is just a single multiplication, this means that the overall cost of the bit decomposition is *three multiplications* per bit (two majority computations and one multiplication for computing d_j).

We now show how to improve this to *two multiplications* per bit instead of three. The idea here is to not explicitly compute the two carry bits c_j, d_j, and instead to leave them implicit in the $\alpha_j, \beta_j, \gamma_j$ values. Specifically, we will show that $\beta_j + \alpha_j$ (with the sum over the integers) actually equals the sum of the carry.5

The actual computation is as follows. As above, compute $[\alpha_j]_1 = [x_1^j]_1 \oplus [x_2^j]_1 \oplus [x_3^j]_1$ and $[\beta_j]_1 = \mathsf{majority}\left(x_1^j, x_2^j, x_3^j\right)$. However, differently to above, compute $[\gamma_j]_1 = \mathsf{majority}\left(\alpha_j, \beta_{j-1}, \gamma_{j-1}\right)$. Finally, compute $[x^j]_1 = [\alpha_j]_1 \oplus [\beta_{j-1}]_1 \oplus [\gamma_{j-1}]_1$.

Proof of correctness. We prove that this is correct by induction. The inductive claim is that for every j, the bit x^j is the correct jth bit of the sum, and the value $\beta_j + \gamma_j \in \{0, 1, 2\}$ with the sum computed *over the integers*, is the carry from the sum $x_1^j + x_2^j + x_3^j + \beta_{j-1} + \gamma_{j-1}$. For $j = 1$, this is trivially the case, since $\beta_0 = \gamma_0 = 0$ and so the bit $x^1 = \alpha_1 = x_1^1 \oplus x_2^1 \oplus x_3^1$, and the carry is just $\mathsf{majority}(x_1^1, x_2^1, x_3^1)$. Assume now that this holds for $j - 1$, and we prove for j. We prove the correctness of this inductive step via a truth table (as above, the computation is symmetric and so it only matter how many ones there are amongst x_1^j, x_2^j, x_3^j, and the value of $\beta_{j-1} + \gamma_{j-1}$).

x_1^j	x_2^j	x_3^j	β_{j-1}	γ_{j-1}	α_j	β_j	γ_j	x^j	**Carry** $\beta_j + \gamma_j$
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	1	0
1	1	0	0	0	0	1	0	0	1
1	1	1	0	0	1	1	0	1	1
0	0	0	1	0	0	0	0	1	0
1	0	0	1	0	1	0	1	0	1
1	1	0	1	0	0	1	0	1	1
1	1	1	1	0	1	1	1	0	2
0	0	0	1	1	0	0	1	0	1
1	0	0	1	1	1	0	1	1	1
1	1	0	1	1	0	1	1	0	2
1	1	1	1	1	1	1	1	1	2

In order to see that this is correct, observe that $x_1^j + x_2^j + x_3^j + \beta_{j-1} + \gamma_{j-1}$ should equal $x^j + 2 \cdot (\beta_j + \gamma_j)$, with all addition over the integers (note that the carry is multiplied by 2 since it is moved to the next bit). By observation, one can verify that this indeed holds for each row. We therefore conclude that the above method correctly computes the sum $[x^1]_1, \ldots, [x^n]_1$ requiring *only two multiplications per bit.*

4.4 Bit Recomposition

In this section, we show how to compute $[x]_n$ from $[x^1]_1, \ldots, [x^n]_n$, where $x = x^n \cdots x^1$ (or stated differently, where $x = \sum_{j=1}^n 2^{j-1} \cdot x^j$). At first sight, it may seem that it is possible for each party to simply locally compute $[x]_n = \sum_{j=1}^n 2^{j-1} \cdot [x^j]_1$, requiring only local scalar multiplications and additions. However, this does not work since the shares $[x^j]_1$ are of bits and not of ring elements, and due to carries one cannot relate to each bit separately and naively embed the bits into ring elements.

We use a similar method to that of bit decomposition, but in reverse order. As in decomposition, there are two steps: local decomposition of the bit shares into three different shares, and then adding with carry. The difference here is that we need to *cancel* out the carry, rather than compute it as in decomposition. In order to see why, assume that we wish to compose two bit sharings into a single sharing in $\mathbb{Z}_{2^2}$. Let $x = 2$ be the value to be composed, and so the parties hold shares of 0 (for the least significant bit) and of 1. Assume now that the sharing of 0 is defined by $x_1^1 = 1$, $x_2^1 = 1$, and $x_3^1 = 0$ (and so $x_1^1 \oplus x_2^1 \oplus x_3^1 = 0$). Then, the sum in the ring of these three shares is actually 2. Thus, this value of 2 in the first bit needs to be *cancelled out* in the second bit. This is achieved by subtracting 1 (or XORing 1) from the second bit. To make this more clear, denote by $\mathsf{bit}(x^j)$ the jth shared bit (as a bit), and by $\mathsf{carry}(x^j)$ the integer carry of the integer-sum of the bit-shares of x^j. For example, if $x_1^j = 1$, $x_2^j = 1$ and $x_3^j = 0$ then $\mathsf{bit}(x^j) = 0$ and $\mathsf{carry}(x^j) = 1$ (the carry equals 1 and not 2, since we consider it as a carry and so it is moved to the left by one bit; stated differently, $x_1^j + x_2^j + x_3^j = \mathsf{bit}(x^j) + 2 \cdot \mathsf{carry}(x^j)$ where addition here is over the integers). Our protocol for bit recomposition works by having the parties in the jth step compute shares in the ring of the value $x^j = \mathsf{bit}(x^j) + 2 \cdot \mathsf{carry}(x^j) - \mathsf{carry}(x^{j-1})$. Finally, they all locally compute $[x]_n = \sum_{j=1}^n 2^{j-1} \cdot [x^j]_n$. This is correct since

$$\begin{aligned}
\sum_{j=1}^{n} 2^{j-1} \cdot [x^j]_n &= \sum_{j=1}^{n} 2^{j-1} \cdot \left(\mathsf{bit}(x^j) + 2 \cdot \mathsf{carry}(x^j) - \mathsf{carry}(x^{j-1})\right) \\
&= \sum_{j=1}^{n} 2^{j-1} \cdot \mathsf{bit}(x^j) + \sum_{j=1}^{n} 2^{j-1} \cdot 2 \cdot \mathsf{carry}(x^j) - \sum_{j=1}^{n} 2^{j-1} \cdot \mathsf{carry}(x^{j-1}) \\
&= \sum_{j=1}^{n} 2^{j-1} \cdot \mathsf{bit}(x^j) + \sum_{j=1}^{n} 2^{j} \cdot \mathsf{carry}(x^j) - \sum_{j=0}^{n-1} 2^{j} \cdot \mathsf{carry}(x^j) \\
&= [x]_n + 2^n \cdot \mathsf{carry}(x^n) - \mathsf{carry}(x^0) = [x]_n
\end{aligned}$$

where the third equality is by simply changing the range of the index j in the third term (from $1, \ldots, n$ to $0, \ldots, n-1$), and the last equality is due to the fact that in the ring $\mathbb{Z}_{2^n}$ the carry to the $(n+1)$th place is just ignored (and that $\mathsf{carry}(x^0) = 0$).

We now describe the algorithm. Let $[x^1]_1, \ldots, [x^n]_1$ be the input bitwise shares; the output should be $[x]_n = \sum_{j=1}^{n} 2^{j-1} \cdot x^j$.

Step 1 – local decomposition: In this step, the parties locally compute shares of the individual bits of their shares, for each j. Specifically, as above, let the sharing $[x^j]_1$ be with values (x_1^j, x_3^j), (x_2^j, x_1^j) and (x_3^j, x_2^j). The parties generate *shares of their shares* as follows, via local computation only:

	$\boldsymbol{P_1}$	$\boldsymbol{P_2}$	$\boldsymbol{P_3}$
Original shares of $\boldsymbol{x^j}$:	(x_1^j, x_3^j)	(x_2^j, x_1^j)	(x_3^j, x_2^j)
New sharing of $\boldsymbol{x_1^j}$:	$(x_1^j, 0)$	$(0, x_1^j)$	$(0, 0)$
New sharing of $\boldsymbol{x_2^j}$:	$(0, 0)$	$(x_2^j, 0)$	$(0, x_2^j)$
New sharing of $\boldsymbol{x_3^j}$:	$(0, x_3^j)$	$(0, 0)$	$(x_3^j, 0)$

At this point, the parties hold $[x_1^j]_1, [x_2^j]_1, [x_3^j]_1$ for $j = 1, \ldots, n$.

Step 2 – add while removing carry: For $j = 1, \ldots, n$, the parties compute the shares $[\alpha_j]_1 = [x_1^j]_1 \oplus [x_2^j]_1 \oplus [x_3^j]_1$, $[\beta_j]_1 = \mathsf{majority}\left(x_1^j, x_2^j, x_3^j\right)$ and $[\gamma_j]_1 = \mathsf{majority}\left(\alpha_j, \beta_{j-1}, \gamma_{j-1}\right)$, where $\beta_0 = \gamma_0 = 0$. (Recall that each majority computation requires one bit-wise multiplication.) Then, the jth bit of the result is mapped to a share $[x^j]_n$ of a ring element by computing

$$[v^j]_1 = [\alpha_j]_1 \oplus [\beta_{j-1}]_1 \oplus [\gamma_{j-1}]_1 \tag{1}$$

and *projecting* the result into the ring. That is, if a party holds a pair of bits $(0, 1)$ for its share of $[v^j]_1$, then it defines $[x^j]_n$ to simply be $(0, 1)$ in the ring $\mathbb{Z}_{2^n}$ (i.e., the integers 0 and 1). Finally, the parties use local computation to obtain $[x]_n = \sum_{j=1}^{n} 2^{j-1} \cdot [x^j]_n$.

We stress that one should not confuse $[v^j]_1$ and $[x^j]_n$; they are both shares of the same value in some sense, *but actually define very different values.* To clarify this, observe that if $v_1^j = v_2^j = 1$ and $v_3^j = 0$, then (v_j^1, v_j^2, v_j^3) constitute a bit sharing of $[v^j]_1 = 0$. However, after projecting this into the ring, we have that it defines a ring-sharing of $[x^j]_n = 2$ (because $v_1^j + v_2^j + v_3^j = 0 \bmod 2$ but $v_1^j + v_2^j + v_3^j = 2 \bmod 2^n$).

Correctness: Correctness of the recomposition procedure is proven in a similar way to the decomposition.

4.5 Reducing the Round Complexity

It is possible to use known methods for adding in $\log n$ rounds, in order to reduce the round complexity of the bit decomposition. However, these come at a cost of much higher AND complexity. Instead, we utilize specific properties of our bit decomposition method in order to reduce the number of rounds, while only mildly raising the number of ANDs. Our method is basically a variation of a *carry-select adder* [1], modified to be suited for bit decomposition. Observe that since the computation is essentially the same for bit decomposition and recomposition (regarding the computation of $\alpha_j, \beta_j, \gamma_j$), the same method here works for recomposition as well.

Recall that bit decomposition works by computing $[\alpha_j]_1 = [x_1^j]_1 \oplus [x_2^j]_1 \oplus [x_3^j]_1$, $[\beta_j]_1 = \mathsf{majority}\left(x_1^j, x_2^j, x_3^j\right)$, and $[\gamma_j]_1 = \mathsf{majority}\left(\alpha_j, \beta_{j-1}, \gamma_{j-1}\right)$. The final shares are obtained by XORing these values and so does not add any additional rounds of communication. Observe that the α_j and β_j shares can all be computed in parallel in a single round. However, the γ_j values must be computed sequentially, since γ_j depends on γ_{j-1}. In order to explain the basic idea behind our tradeoff between computation and rounds, we show concretely how to reduce the number of rounds to approximately one half and one quarter, and then explain the general tradeoff:

Reducing to $n/2+2$ rounds. As described above, all of the α_j, β_j values can be computed in the first round, at the cost of exactly n AND gates. Next, the parties compute the following:

1. $\gamma_1, \ldots, \gamma_{n/2}$ at the cost of $n/2$ rounds and $n/2$ AND gates,
2. $\gamma_{n/2+1}, \ldots, \gamma_n$ under the *assumption* that $\gamma_{n/2} = 0$, at the cost of $n/2$ rounds and $n/2$ AND gates, and
3. $\gamma_{n/2+1}, \ldots, \gamma_n$ under the *assumption* that $\gamma_{n/2} = 1$, at the cost of $n/2$ rounds and $n/2$ AND gates.

Observe that all three computations above can be carried out in parallel, and thus this requires $n/2$ rounds overall. Next, the parties use a MUX to compute which $\gamma_{n/2+1}, \ldots, \gamma_n$ values to take; this is possible since $\gamma_{n/2}$ is already known at this point. This MUX uses a single AND gate per bit, coming to a total of $n/2$ AND gates, and a single round. The overall cost is $3n$ AND gates and $n/2+2$ rounds. Concretely for 32 bit values, this results in 96 AND gates and 17 rounds (instead of 64 AND gates and 32 rounds).

Reducing to $n/4+4$ rounds. This time we divide the γ_j values to be computed into 4 parts, as follows. In the first round, all α_j, β_j values are computed. Then:

1. $\gamma_1, \ldots, \gamma_{n/4}$ are computed at the cost of $n/4$ rounds and $n/4$ AND gates,
2. In parallel to the above, $\gamma_{\frac{i \cdot n}{4}+1}, \ldots, \gamma_{\frac{(i+1)\cdot n}{4}}$ for $i = 1, 2, 3$ are computed all in parallel, each under the *assumption* that $\gamma_{\frac{i \cdot n}{4}} = 0$ and that $\gamma_{\frac{i \cdot n}{4}} = 1$, at the cost of $n/4$ rounds and $n/4$ AND gates each (overall 6 such computations).

When all of the above are completed, the parties compute *sequentially* the MUX over $\gamma_{\frac{i\cdot n}{4}+1}, \ldots, \gamma_{\frac{(i+1)\cdot n}{4}}$ given $\gamma_{\frac{i\cdot n}{4}}$ for $i = 1, 2, 3$ (each at a cost of $n/4$ AND gates and 1 round). The overall cost is $3.5n$ AND gates and $n/4 + 4$ rounds. Concretely for 32 bit values, this results in 112 AND gates and 12 rounds (instead of 64 AND gates and 32 rounds).

The general case. The above method can be used to divide the γ_j values into ℓ blocks. In this case, the number of rounds is $\frac{n}{\ell}$ to compute the γ values, and $\ell - 1$ to compute the sequential MUXes. Using a similar computation to above, we have that the overall number of rounds is $\frac{n}{\ell} + \ell$, and the number of AND gates is $n + \frac{(3\ell-2)n}{\ell}$. With this method, the number of rounds is minimized when $\frac{n}{\ell} = \ell$, which holds when $\ell = \sqrt{n}$ and results in $2\sqrt{n}$ rounds. In this case, the number of AND gates to be computed equals $4n - 2\sqrt{n}$. Importantly, this method provides a tradeoff between the number of rounds and the number of AND gates, since less blocks means less MUX computations. See Table 1 for a comparison on the number of rounds and AND gates, minimizing the number of rounds and minimizing the number of AND gates, when using our method. (Note that the minimum number of AND gates is always obtained by taking $\ell = 1$; i.e., by using the original method above.) These values are computed using the general equations above.

	Minimal ANDs		**Minimal Rounds**		
Size n	**ANDs**	**Rounds**	$\boldsymbol{\ell}$	**ANDs**	**Rounds**
16	32	16	4	56	8
32	64	32	4	112	12
64	128	64	8	240	16
128	256	128	10	487	23

Table 1: Different parameters and their cost

Somewhat surprisingly, it is possible to do even better by using a *variable-length carry-adder* approach. The idea behind this is that it is possible to start computing the MUXes for the first blocks while still computing the γ values for the later blocks. To see why this is possible, consider the concrete example of $n = 16$ and $\ell = 4$. When dividing into equal-size blocks of length 4, the overall number of rounds is 8 (1 round for computing α_j, β_j and 7 for the rest). For this concrete case, we could divide the input into *five* blocks of sizes 2,2,3,4,5, respectively. Observe that the MUX needed using the result of the first block to choose between the two results of the second block can be computed in parallel to the last γ value on the third block. Likewise, the next MUX can be computed in parallel to the last γ value of the fourth block, and so on. In this way, there are no wasted rounds, and the overall number of rounds is reduced from 7 to 6. Although this is a modest improvement, for larger values of n, it is more significant. For example, we need 18 rounds for bit decomposition of 128-bit values, in contrast to 23 rounds with fixed-length blocks (see Table 1). We wrote a script to find the optimal division into blocks for this method, for various values of n; the results appear in Table 2.

Observe that the number of ANDs required for this method is greater than in Table 1, thus further contributing to the aforementioned tradeoff. We stress that this tradeoff is significant since different parameters may provide better performance on slow, medium and fast networks.

Size n	**ANDs**	**Rounds**	**Block Sizes**
16	63	7	$5, 4, 3, 2, 2$
32	128	10	$8, 7, 6, 5, 4, 2$
64	255	13	$11, 10, 9, 8, 7, 6, 5, 4, 4$
128	519	18	$16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 2$

Table 2: Optimal block-size and costs for the variable-length approach (computation is from right-to-left)

Bit decomposition using conditional sum adders. We conclude with a different approach that is based on a conditional sum adder. This variant takes a divide-and-conquer approach to computing the blocks. That is, it splits the n-bit input into two blocks of $n/2$ bits, uses a conditional sum adder to compute the sum of the lower block with carry 0 and the sum of the higher block with carries 0 and 1, and then uses MUX gates to select the correct outputs for the higher block. At the bottom level, a pair of bits is simply added using a full adder. This tree-based approach leads to a logarithmic number of rounds at an overall cost of $O(n \log n)$ AND gates, since there are a linear number of MUX gates at every level. The concrete costs for this method are presented in Table 3. As can be seen, the number of rounds is significantly reduced, but at the cost of a notable increase in the number of ANDs. In slow networks with fast computing devices, this approach can be preferable.

Size n	**ANDs**	**Rounds**
16	121	6
32	280	7
64	631	8
128	1398	9

Table 3: Costs for the conditional-sum adder approach

4.6 Security

Let v be a value. We say that $\mathsf{type}(v) = \mathbb{Z}_{2^n}$ if $v \in \mathbb{Z}_{2^n}$ and we say that $\mathsf{type}(v) = \mathbb{Z}_2$ if $v \in \{0, 1\}$. We will relate to the addition, scalar multiplication and multiplication of values below. In all cases, these operations are only possible for values of the *same type*.

In $\mathcal{F}_{\text{mpc}}$, we define a general MPC functionality that enables carrying out standard operations on shared values: addition, scalar multiplication and multiplication (beyond sharing input and getting output). However, in contrast to the usual definition, we define $\mathcal{F}_{\text{mpc}}$ to carry out these operations on both shares of bits and shares of ring elements. In addition, the functionality enables the decomposition of a ring element in $\mathbb{Z}_{2^n}$ to n shares of bits, and the recomposition of n shares of bits to a ring element. This provides a much more general functionality since computations can be carried out both using arithmetic circuits and Boolean circuits.

Observe that add and $\mathsf{scalarmult}$ in $\mathcal{F}_{\text{mpc}}$ are operations that depend only on the honest parties. This is due to the fact that they involve local operations only, and thus the adversary cannot interfere in their computation. The standard $\mathcal{F}_{\text{mpc}}$ functionality fulfilled by secret-sharing based protocols is the same as Functionality 4.1, with the exception that all operations of one type only

FUNCTIONALITY 4.1 (The Mixed MPC Functionality $\mathcal{F}_{\mathrm{mpc}}$)

$\mathcal{F}_{\mathrm{mpc}}$ runs with parties $P_1, \ldots, P_m$ and the ring $\mathbb{Z}_{2^n}$, as follows:

- Upon receiving $(\mathsf{input}, id, i, v)$ from party P_i where $v \in \mathbb{Z}_{2^n}$ or $v \in \{0,1\}$ and id has not been used before, $\mathcal{F}_{\mathrm{mpc}}$ sends (input, id, i) to all parties and locally stores (id, v).
- Upon receiving $(\mathsf{add}, id_1, id_2, id_3)$ from all *honest* parties, if there exist v_1, v_2 such that (id_1, v_1) and (id_2, v_2) have been stored and $\mathsf{type}(v_1) = \mathsf{type}(v_2)$, and id_3 has not been used before, then $\mathcal{F}_{\mathrm{mpc}}$ locally stores $(id_3, v_1 + v_2)$.
- Upon receiving $(\mathsf{scalarmult}, id_1, id_2, c)$ from all *honest* parties, if there exists a v such that (id_1, v) has been stored and $\mathsf{type}(c) = \mathsf{type}(v)$, and id_2 has not been used before, then $\mathcal{F}_{\mathrm{mpc}}$ locally stores $(id_2, c \cdot v)$.
- Upon receiving $(\mathsf{mult}, id_1, id_2, id_3)$ from all parties, if there exist v_1, v_2 such that (id_1, v_1) and (id_2, v_2) have been stored and $\mathsf{type}(v_1) = \mathsf{type}(v_2)$, and id_3 has not been used before, then $\mathcal{F}_{\mathrm{mpc}}$ locally stores $(id_3, v_1 \cdot v_2)$.
- Upon receiving $(\mathsf{decompose}, id, id_1, \ldots, id_n)$ from all parties, if there exists a v such that (id, v) has been stored and $\mathsf{type}(v) = \mathbb{Z}_{2^n}$, and $id_1, \ldots, id_n$ have not been used before, then $\mathcal{F}_{\mathrm{mpc}}$ locally stores (id_i, v_i) for $i = 1, \ldots, n$, where $v = v_1, \ldots, v_n$.
- Upon receiving $(\mathsf{recompose}, id_1, \ldots, id_n, id)$ from all parties, if there exist $v_1, \ldots, v_n$ such that (id_i, v_i) has been stored and $\mathsf{type}(v_i) = \mathbb{Z}_2$ for all $i = 1, \ldots, n$, and id has not been used before, then $\mathcal{F}_{\mathrm{mpc}}$ locally stores (id, v), where $v = v_1, \ldots, v_n$.
- Upon receiving (output, id, i) from all parties, if there exists a v such that (id, v) has been stored then $\mathcal{F}_{\mathrm{mpc}}$ sends (output, id, v) to party P_i.

and there are not decompose or recompose operations. We denote the standard $\mathcal{F}_{\mathrm{mpc}}$ functionality that works over the ring R by $\mathcal{F}^{R}_{\mathrm{mpc}}$ (and so denote $\mathcal{F}^{\mathbb{Z}_2}_{\mathrm{mpc}}$ for bits and $\mathcal{F}^{Z_{2^n}}_{\mathrm{mpc}}$ for the ring $\mathbb{Z}_{2^n}$.

Security of bit decomposition and recomposition. The fact that our protocols are secure follow immediately from the fact that they are comprised solely of the following elements:

- Local transformation operations from *valid* bit shares to *valid* ring shares and vice versa,
- Bit-MPC add and multiply operations over bit shares, and
- Ring-MPC add and multiply operations over ring shares.

Since the MPC operations use secure protocols and work on valid shares of the appropriate type, these operations are securely carried out. Furthermore, since the local transformations require no communication, an adversary cannot cheat. Thus, the combination of the bit and ring protocols, along with the bit decomposition and recomposition protocols presented above, constitute a protocol that securely computes the mixed MPC functionality $\mathcal{F}_{\mathrm{mpc}}$.

In order for the above to work, we need the bit and ring protocols to have the property that the original simulator (that did not consider bit-decompositon) also successfully simulate the protocol in the presence of the bit-decomposition protocols. It is straightforward to see that the simulation of the new protocol is exactly the same as before except for outputs that depend on a share of some value. This happens when local bit decomposition converts a share into a secret shared value. This simulation is not trivial since the above MPC functionality does not keep shares as its internal state. Nevertheless, fortunately, this exception does not happen in our functionality since full bit-decomposition does not reveal any value that depends on a share. By being deliberate in this point, we are able to simulate the functionality as before.

4.7 Theoretical Efficiency

The complexity of our MPC conversions of shares between that of $\mathbb{Z}_{2^n}$ and that of $\mathbb{Z}_2^n$ are given in Table 4. These numbers refer to the three-party protocol of [3] that is secure in the presence of malicious adversaries. The most optimized version of that protocol requires each party to send just 7 bits per AND gate, meaning an overall cost of 21 bits per AND gate for all parties. In Table 4 we provide the communication cost and number of rounds for our protocol, with different tradeoffs between computation and round complexity:

Version	**Total comm. bits**	**Rounds**
Basic conversion ($n = 32$)	1,344	32
Variable-length adder ($n = 32$)	2,688	10
Conditional-sum adder ($n = 32$)	5,880	7
Basic conversion ($n = 128$)	5,376	128
Variable-length adder ($n = 128$)	10,899	18
Conditional-sum adder ($n = 128$)	29,358	9

Table 4: Complexity of decomposition and recomposition

We now compare our protocol to the previous best protocols. We stress that previous protocols work generically for *any* ring, and as such are more general. However, this shows that much can be gained by focusing on rings of specific interest, especially the ring of integers which is of interest in many real-world computations. In Table 5, we present the cost of our protocols versus those of [12], [27] and [30], when applied to the ring $\mathbb{Z}_{2^{32}}$. In all cases, we consider the concrete cost when using the low-communication three-party protocol of [3] that requires only 7-bits of communication per party per AND gate. The results show the striking improvement our method makes over previous protocols, for the *specific case* of the ring $\mathbb{Z}_{2^n}$, and when using replicated secret sharing. For our protocol, we present the costs for the 32-round version, with minimum AND complexity.

Protocol	**Method**	**Total comm. bits**	**Rounds**
Bit-Decomposition	[12] +[3]	723,912	38
	[27] +[3]	408,072	25
	Ours+[3]	1,302	32
Ring-Composition	[30] +[3]	340, 704	62
	Ours+[3]	1,302	32

Table 5: Comparison of Complexity of 32-bit Integer Conversions for Secure 3-Party Computation

We remark that a direct conversion of the SPDZ bit decomposition method to the case of rings would yield the complexity of [12]+[3] in Table 5. Thus, our special-purpose conversion protocols are significant with respect to the efficiency of the result.

5 Experimental Evaluation

In order to evaluate our toolchain and protocol, we have implemented various computations, ranging from a simple mean and variance computations, to a more involved computations of inference via a non-balanced decision tree and the private processing of an SQL query. The SQL query is quite a complex computation and is derived from the following query for a typical survey:

```
SELECT count(*), avg(credit limit) FROM Census
WHERE State==Utah
GROUP BY Age, Sex HAVING count(*) > 100;
```

This query computes the average credit limit of every age-group and sex (i.e., average credit limit of 30 year old females, average credit limit of 30 year old males, and so on), outputting only results for sets that have at least 100 data items in the set. This last requirement is necessary to preserve privacy and to ensure that there are no results based on very few individuals. For all fields that have a small range such as state, age, and sex we input the data in a bit-wise unary encoding (a list of bits of which only one is 1), which simplifies the selection operation in secure computation.

The decision tree private inference example uses the decision tree built from real data published for a paper on credit decisions [28]. The concrete decision tree in our computation has 1256 leaves at depths from 4 to 30. Since multiparty computation reveals the amount of computation (i.e., how many gates are computed), we have to always execute 30 decisions in order to hide the path traversed in the evaluation. This is achieved using dummy data if a leaf is reached before the last step. Furthermore, traversal of the tree makes use of oblivious selection from the current depth of the tree represented as an array (this selection is of the node to be used in the current level of the decision tree), in order to not reveal anything about the path of computation in the tree.

Whenever non-integer computation is required, we use fixed-point computation as implemented in the SPDZ compiler [7]. This is justified because the mean over a set of numbers in a limited range will also be in this range and thus not require the larger range of floating-point numbers. The bit decomposition and recomposition used for the SQL query is the SPDZ compiler method for SPDZ and MHM$\mathbb{Z}_p$, and is our new method from Section 4. The times given are for the basic conversion (see Table 4) which minimizes the amount of communication at the expense of a higher number of rounds. (We also implemented the other versions, but they were slower in our tests since we ran the experiments on a very low-latency network.)

We ran our experiments on AWS with three parties in a single region, using `m5.12xlarge` instances providing 10 Gbps network communication. The only exception is for the BMR protocol, where we used `i3.2xlarge` instances due to the increased amount of storage needed to store the garbled circuit.

Figures 6–8 show the online times for mean, variance, and our SQL query for various numbers of inputs, and Table 6 shows the results of decision tree computation. MHM $\mathbb{Z}_p$ refers to the malicous honest-majority protocol over $\mathbb{Z}_p$ of [24], while SHM $\mathbb{Z}_{2^n}/\mathbb{Z}_2$ refers to the semi-honest honest-majority protocol of [4] over the ring of integers $\mathbb{Z}_{2^n}$ for any $n \geq 1$. Note that the different protocols operate in different security models: SPDZ and BMR provide security in the presence of any $t \leq n$ malicious corruptions, MHM $\mathbb{Z}_p$ provides security in the presence of a malicious minority, and SHM $\mathbb{Z}_{2^n}/\mathbb{Z}_2$ provides security in the presence of semi-honest adversaries with an honest minority. This explains the expensive offline phase for SPDZ and BMR because more expensive operations such as somewhat homomorphic encryption are used there. To time the offline phase of SPDZ, the newer "Low Gear" protocol [20] has been used on `r4.8xlarge` instances due to the larger memory requirement of homomorphic encryption, while the MASCOT protocol [19] has been used for BMR. (The SPDZ offline was also computed using a large number of threads, in contrast to a single thread for MHM/SHM.)

We stress that we present these results to demonstrate the new capability of writing a single complex program and running it on *four completely different low-level protocols*, and not in order to compare efficiency. Indeed, we are continuing our work to improve the efficiency of the SPDZ-compiled lower-level protocols (e.g., adding vectorization, more parallelism and specific optimizations). Nevertheless, it is interesting to observe that the SHM $\mathbb{Z}_{2^n}$ method is approximately *50 times faster* than the MHM $\mathbb{Z}_p$ method for the SQL processing (Figure 8). Although there is a difference between semi-honest and malicious, the cost of MHM $\mathbb{Z}_{2^n}$ is only 7-times slower than SHM $\mathbb{Z}_{2^n}$ [3]. The rest of the difference is due to the faster bit decomposition and recomposition for the ring-based protocol versus the field-based protocol.

	Resources	**SPDZ**	**MHM $\mathbb{Z}_p$**	**SHM $\mathbb{Z}_{2^n}$**	**BMR**
Security level:		Malicious $t \leq n$	Malicious $t < n/2$	Semi-honest $t < n/2$	Malicious $t \leq n$
Online: time:	1 core	0.3005	3.0416	0.4641	0.5353
# rounds:		783	584	2746	28
Offline: time:	48 cores	5.2204	Not required	Not required	1041.8

Table 6: Decision tree computation (seconds).

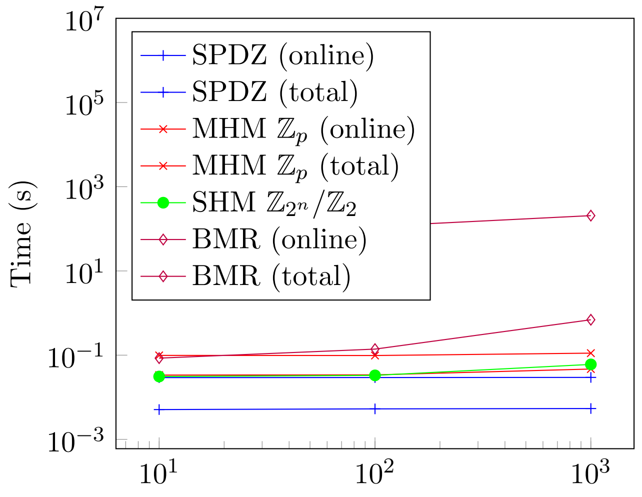


Figure 6: Mean computation (X-axis=num. inputs).

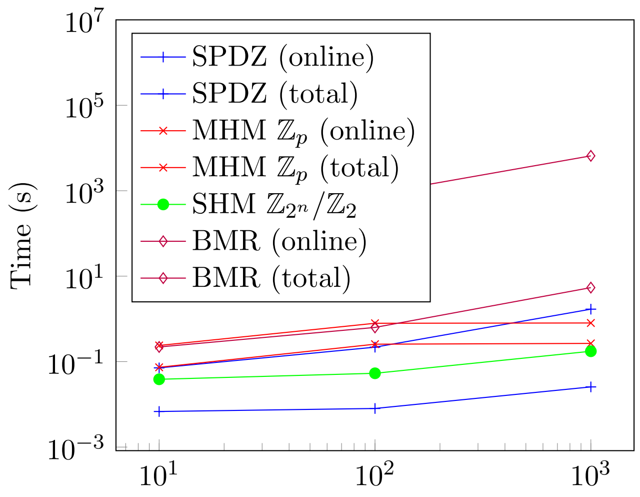


Figure 7: Variance computation (X-axis=num. inputs).

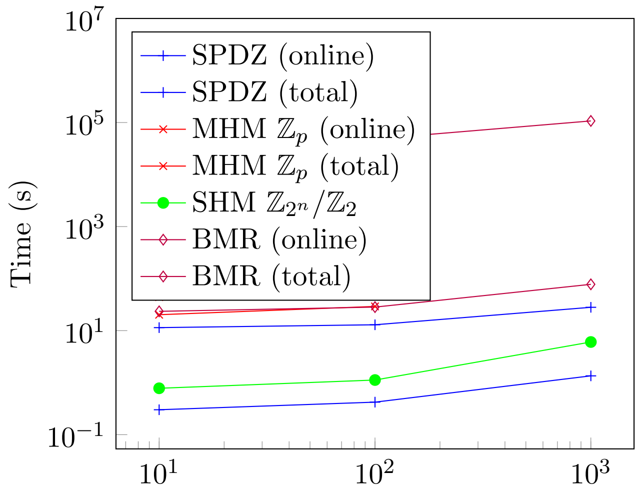


Figure 8: US Census SQL query (X-axis=num. inputs).

Batch vectorization. We have implemented batch vectorization for the Ring-based protocol at the VM level. This works by defining the level of vectorization desired, and then the same single-execution code written at the compiler level is run on vectors of the specified length. For example, defining vectorization of level 64 for the decision tree inference problem means that inference is run on 64 inputs at the same time. This works by representing each element as a vector of 64, and running the MPC in parallel for each.

We ran these batch executions on the same problems as above; these results appear in Table 7. Observe that the "non-batch" and "Batch $\times$ 1" both run a single execution, but there is a fixed overhead in the VM for running the batched experiments. Comparing these two columns, one can see that this overhead is quite high; we are working on reducing it. Beyond this, observe that the cost of batching many executions together is very minor. Thus, a single decision tree inference (without batching) takes approximately 0.5 seconds whereas 64 in parallel takes just under 6 seconds, or an average of under 0.1 second. We believe that by reducing the fixed overhead, we will obtain that parallelism is essentially for free. This is of great importance in many real-world use cases where the same computation is carried out many times. For example, census statistics like the SQL query in our example would be computed for every state, and so could be vectorized.

	Non-batch	**Batch** $\times$ 1	**Batch** $\times$ 8	**Batch** $\times$ 32	**Batch** $\times$ 64
Mean (10 inputs)	0.031	0.139	0.138	0.136	0.149
Mean (100 inputs)	0.033	0.145	0.145	0.142	0.153
Mean (1000 inputs)	0.060	0.178	0.184	0.176	0.171
Variance (10 inputs)	0.039	0.362	0.371	0.391	0.381
Variance (100 inputs)	0.053	0.428	0.688	0.677	0.687
Variance (1000 inputs)	0.175	2.501	2.318	2.348	2.461
SQL (10 inputs)	0.779	10.233	10.335	10.997	11.285
SQL (100 inputs)	1.122	10.766	11.029	11.754	13.606
SQL (1000 inputs)	6.039	17.755	15.216	31.154	36.471
Decision tree	0.464	2.949	3.276	4.399	5.945

Table 7: Running times for batch vectorization in seconds. Batch $\times$ N means running N executions in parallel (i.e., with vectors of length N).

Open source. Our code is open source and available for free use. Our fork of SPDZ-2, including our extensions and hooks to them and changes to the compiler to support adding instructions and so on, can be found at https://github.com/cryptobiu/SPDZ-2. Furthermore, the extension required for plugging in the multiparty honest-majority protocol of [24] can be found at https://github.com/cryptobiu/SPDZ-2-Extension-MpcHonestMajority.

6 Future Work

This paper describes the first steps towards making the SPDZ compiler a general-purpose tool that can enable the use of MPC by software developers without MPC expertise. In order to complete this task, more work is needed in the following areas:

- *Efficiency:* The current run-time requires additional optimizations to achieve running-time that is comparable to that of a native protocol that works directly with a circuit and is optimized for latency or throughput. It is unreasonable to assume that a general compiler will achieve the same level of efficiency as a tailored optimized version of a protocol. Nevertheless, the usability gains are significant enough so that a reasonable penalty (of say, 15%) is justified. An important goal is thus to achieve efficiency of this level, and we are currently working on this.

- *Protocol generality:* As we have argued, there is no single MPC protocol that is best for every task. On the contrary, we now understand that many different protocols of different types are needed for different settings. The best protocol depends on the efficiency goal (low latency or high throughput), the network setting (LAN or WAN), the function being computed (arithmetic or Boolean or mixed, and if mixed how many transitions are needed), and so on. In order to achieve this goal, more protocols need to be incorporated into the SPDZ compiler framework.

In addition to the above, we believe that an additional method should be added that outputs a circuit (arithmetic, Boolean or mixed) generated from the Python code. This deviates from the SPDZ run-time paradigm and requires running the protocol with a specific circuit, but it enables the use of the compiler in the more traditional circuit-compiler methodology that also has advantages. In particular, it can be used for protocols that have not been incorporated into the SPDZ run-time, and for optimized code that works specifically with a static circuit.

- *Compiler generality:* The SPDZ compiler is already very general and provides support for a rich high-level language. However, as more real use cases are discovered, it will need to be further enriched. This work is already being done independently on the original SPDZ compiler and we hope that these works will be merged, for the benefit of the general community.

References

[1] Carry-Select Adder, Wikipedia, March 2018. https://en.wikipedia.org/wiki/Carry-select_adder

[2] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure Computation on Floating Point Numbers. In *NDSS 2013*, 2013.

[3] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman and O. Weinstein. Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier. In the *38th IEEE Symposium on Security and Privacy*, pages 843–862, 2017.

[4] T. Araki, J. Furukawa, Y. Lindell, A. Nof and K. Ohara. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In the 23*rd ACM CCS*, pages 805–817, 2016.

[5] Donald Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO'91*, Springer (LNCS 576), pages 420–432, 1992.

[6] D. Beaver, S. Micali, and P. Rogaway. The Round Complexity of Secure Protocols. In the *22nd STOC*, pages 503–513, 1990.

[7] Bristol Cryptography Group. SPDZ software. https://www.cs.bris.ac.uk/Research/CryptographySecurity/SPDZ/, 2016.

[8] Niklas Buscher, Andreas Holzer, Alina Weber and Stefan Katzenbeisser. Compiling Low Depth Circuits for Practical Secure Computation. In *ESORICS 2016*, pages 80–98, 2016.

[9] Niklas Buscher and Stefan Katzenbeisser. *Compilation for Secure Multi-party Computation.* Springer Briefs in Computer Science, Springer 2017.

[10] Octavian Catrina and Sebastiaan de Hoogh. Secure Multiparty Linear Programming Using Fixed-Point Arithmetic. In *ESORICS 2010*, Springer (LNCS 6345), pages 134–150, 2010.

[11] Octavian Catrina and Amitabh Saxena. Secure Computation With Fixed-Point Numbers. In *FC 2010*, Springer (LNCS 6052), pages 35–50, 2010.

[12] I. Damgård, M. Fitzi, E. Kiltz, J.B. Nielsen, and T. Toft. Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation. In the 3rd *Theory of Cryptography Conference* (TCC), Springer (LNCS 3876), pages 285-304, 2006.

[13] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl and N.P. Smart. Practical Covertly Secure MPC for Dishonest Majority - or: Breaking the SPDZ Limits. In 18th *ESORICS*, pages 1–18, 2013.

[14] I. Damgård, V. Pastro, N.P. Smart and S. Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO 2012*, pages 643–662, 2012.

[15] M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart and H. Veith. CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations. In *CC 2014*, pages 244–249, 2014.

[16] J. Furukawa, Y. Lindell, A. Nof and O. Weinstein. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority In *EUROCRYPT 2017*, Springer (LNCS 10211), pages 225–255, 2017.

[17] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low Cost Constant Round MPC Combining BMR and Oblivious Transfer. In *ASIACRYPT 2017* Springer (LNCS 10624), pages 598–628, 2017.

[18] Marcel Keller. The Oblivious Machine - Or: How to Put the C Into MPC. Cryptology ePrint Archive, Report 2015/467, 2015. http://eprint.iacr.org/2015/467.

[19] 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 16*, pages 830–842. ACM Press, October 2016.

[20] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT 2018*, Springer (LNCS 10822), pages 158–189, 2018.

[21] Marcel Keller, Peter Scholl, and Nigel P. Smart. An Architecture for Practical Actively Secure MPC With Dishonest Majority. In *ACM CCS 2013*, pages 549–560, 2013.

[22] Marcel Keller and Avishay Yanai. Efficient Maliciously Secure Multiparty Computation for RAM. In *EUROCRYPT 2018*, Springer (LNCS 10822), pages 91–124, 2018.

[23] Marcel Keller and Avishay Yanay. ORAM in SPDZ-BMR, 2018. https://github.com/mkskeller/SPDZ-BMR-ORAM.

[24] Y. Lindell and A. Nof. A Framework for Constructing Fast MPC over Arithmetic Circuits with Malicious Adversaries and an Honest-Majority. In the 24th *ACM CCS*, pages 259–276, 2017.

[25] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient Constant Round Multi-Party Computation Combining BMR and SPDZ. In *CRYPTO 2015*, Springer (LNCS 9216), pages 319–338, 2015.

[26] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44:519–521, 1985.

[27] T. Nishide and K. Ohta. Multiparty Computation for Interval, Equality, and Comparison Without Bit-Decomposition Protocol. In the 10*th PKC*, Springer (LNCS 4450), pages 343–360, 2007.

[28] Vivek Kumar Singh, Burcin Bozkaya, Alex Pentland, Money Walks: Implicit Mobility Behavior and Financial Well-Being, PLoS ONE 10(8): e0136628. https://doi.org/10.1371/journal.pone.0136628

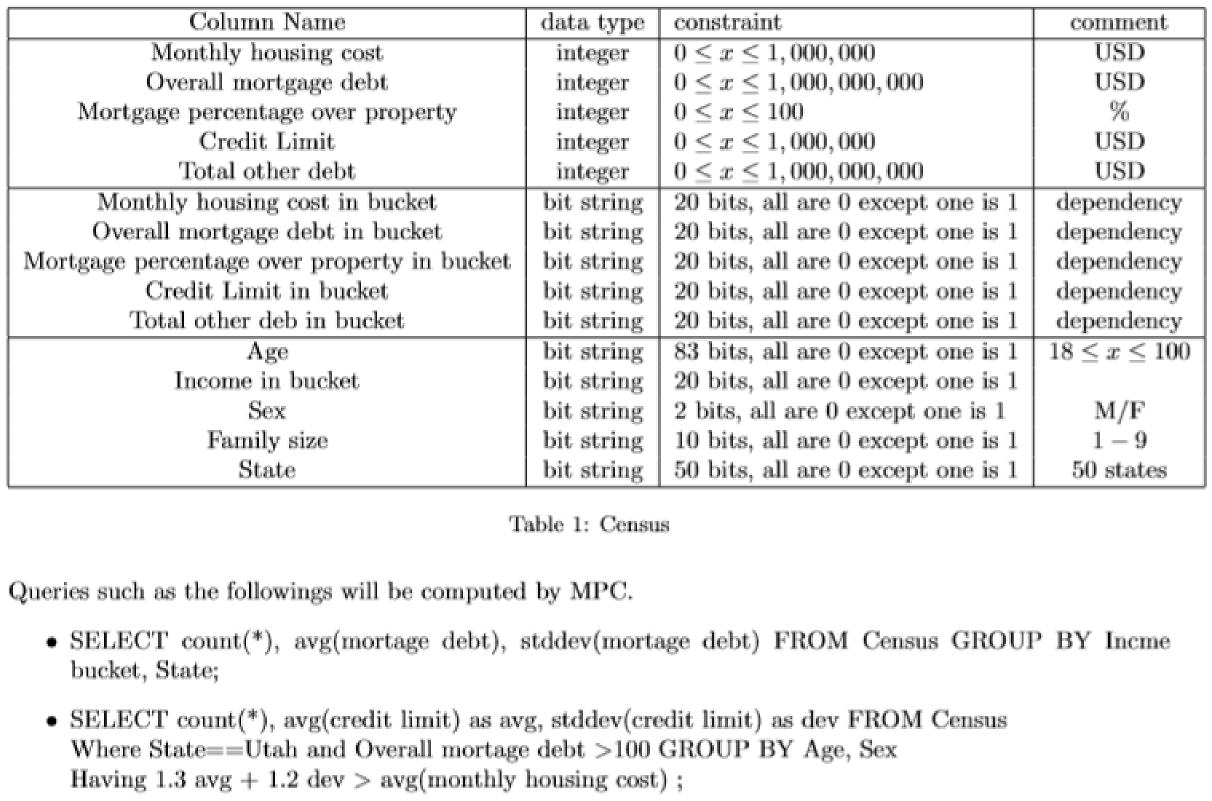
[29] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, Farinaz Koushanfar. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. *IEEE Symposium on Security and Privacy 2015*, pages 411–428, 2015.

[30] T. Toft. Constant-Rounds, Almost-Linear Bit-Decomposition of Secret Shared Values. In *CT-RSA 2009*, Springer (LNCS 5473), pages 357–371, 2009.

[31] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation. In *ACM CCS 17*, pages 21–37, 2017.

A From SQL to Pseudocode to SPDZ

On Thursday April 12, at 8:45am, the minutes of our morning meeting defined an MPC experiment involving a complex SQL query. The experiment was defined according to the table and query appearing in Figure 9. We stress that although the researchers in our lab had already gained some experience in "SPDZ programming", these involved programming a decision tree and genetic computation and so were in completely different domains.



Column Name	data type	constraint	comment
Monthly housing cost	integer	$0 \le x \le 1,000,000$	USD
Overall mortgage debt	integer	$0 \le x \le 1,000,000,000$	USD
Mortgage percentage over property	integer	$0 \le x \le 100$	%
Credit Limit	integer	$0 \le x \le 1,000,000$	USD
Total other debt	integer	$0 \le x \le 1,000,000,000$	USD
Monthly housing cost in bucket	bit string	20 bits, all are 0 except one is 1	dependency
Overall mortgage debt in bucket	bit string	20 bits, all are 0 except one is 1	dependency
Mortgage percentage over property in bucket	bit string	20 bits, all are 0 except one is 1	dependency
Credit Limit in bucket	bit string	20 bits, all are 0 except one is 1	dependency
Total other deb in bucket	bit string	20 bits, all are 0 except one is 1	dependency
Age	bit string	83 bits, all are 0 except one is 1	$18 \le x \le 100$
Income in bucket	bit string	20 bits, all are 0 except one is 1	
Sex	bit string	2 bits, all are 0 except one is 1	M/F
Family size	bit string	10 bits, all are 0 except one is 1	$1-9$
State	bit string	50 bits, all are 0 except one is 1	50 states

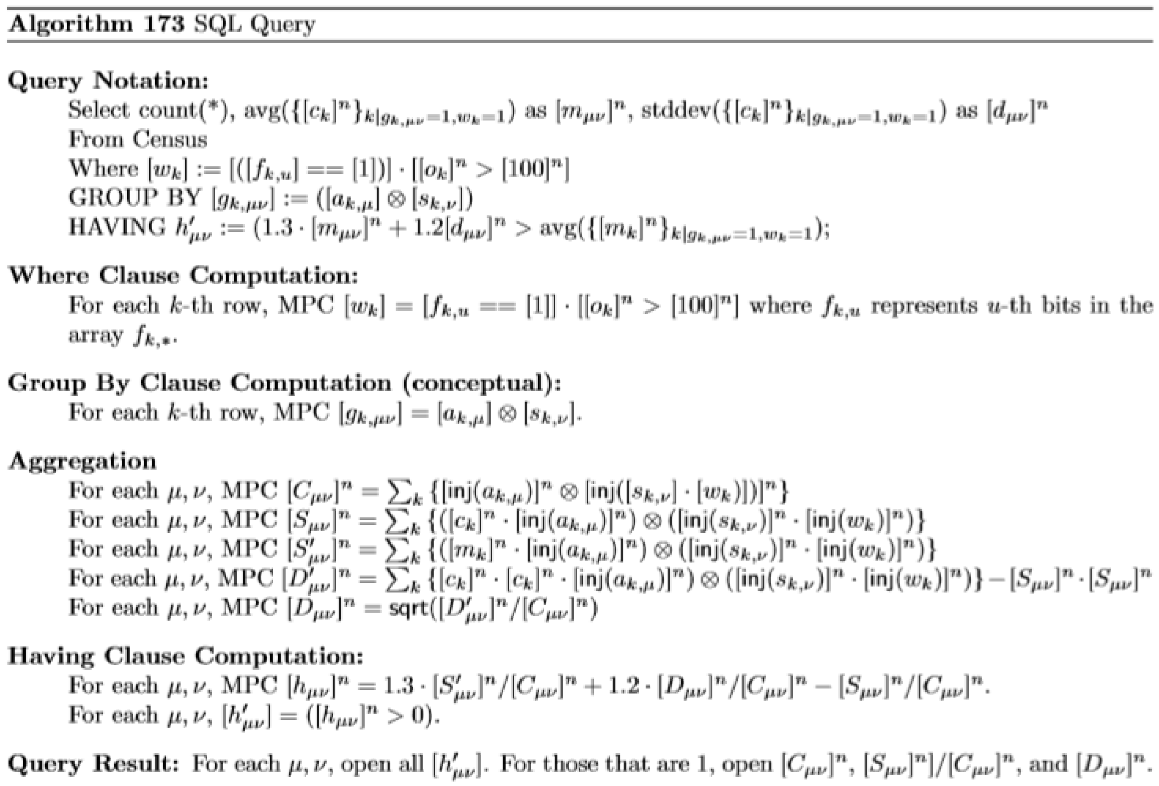
Table 1: Census

Queries such as the followings will be computed by MPC.

- SELECT count(*), avg(mortage debt), stddev(mortage debt) FROM Census GROUP BY Incme bucket, State;
- SELECT count(*), avg(credit limit) as avg, stddev(credit limit) as dev FROM Census
  Where State==Utah and Overall mortage debt >100 GROUP BY Age, Sex
  Having 1.3 avg + 1.2 dev > avg(monthly housing cost) ;

Figure 9: The SQL experiment definition (8:45am).

At 11:16am on the same day, the researchers sent an email specifying the pseudocode for processing the query; see Figure 10. The pseudocode was written based on an understanding of MPC costs and thus how to optimize (e.g., by storing discrete values in bit vectors that are 0 in all places except for the place representing the value – i.e., age can represented by a vector of length 120 and a person of age 40 will have the 40th bit equal 1 and all else equal 0).

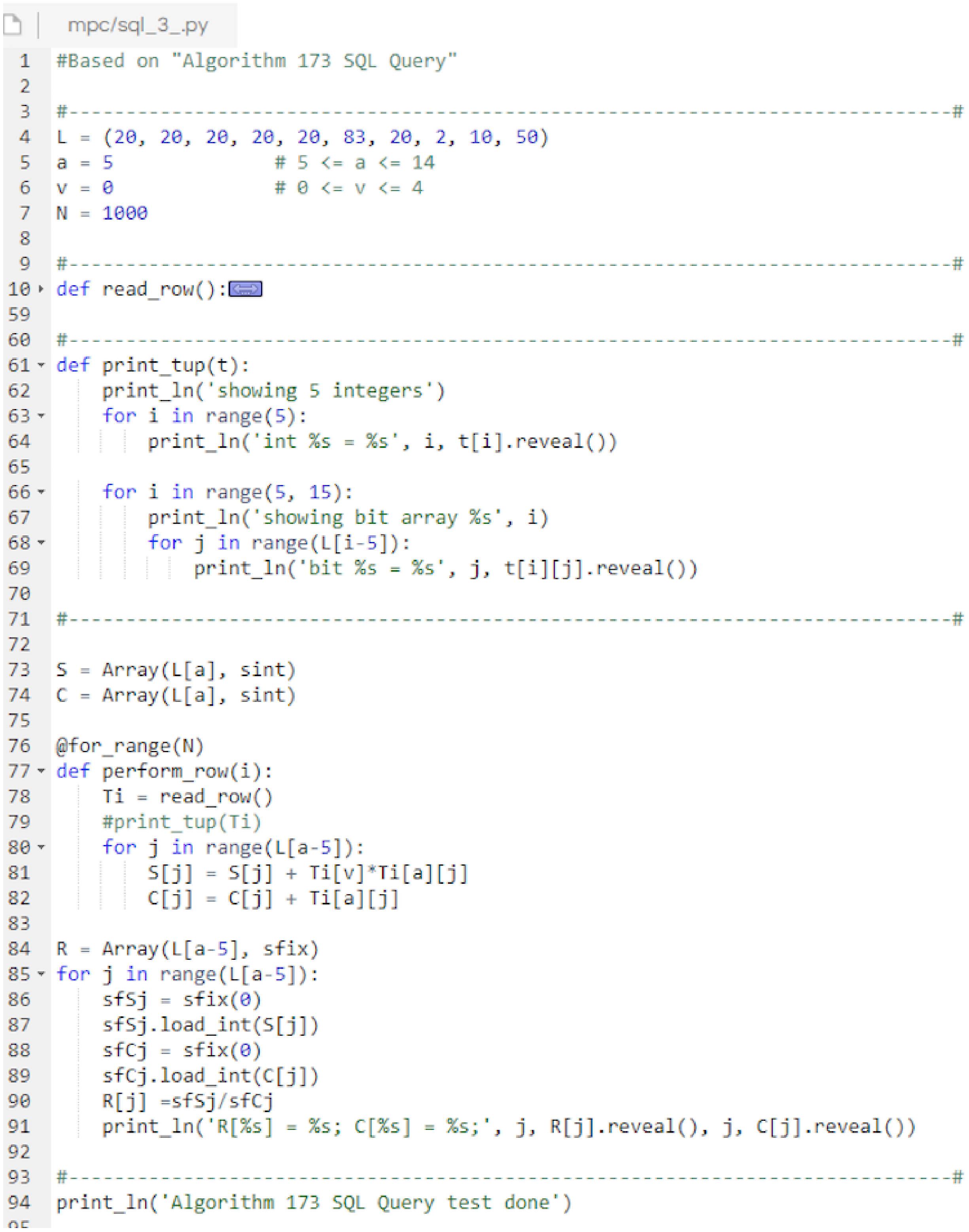


```
Algorithm 173 SQL Query

Query Notation:
    Select count(*), avg($\{[c_k]^n\}_{k|g_{k,\mu\nu}=1,w_k=1}$) as $[m_{\mu\nu}]^n$, stddev($\{[c_k]^n\}_{k|g_{k,\mu\nu}=1,w_k=1}$) as $[d_{\mu\nu}]^n$
    From Census
    Where $[w_k] := [([f_{k,u}] == [1])] \cdot [[o_k]^n > [100]^n]$
    GROUP BY $[g_{k,\mu\nu}] := ([a_{k,\mu}] \otimes [s_{k,\nu}])$
    HAVING $h'_{\mu\nu} := (1.3 \cdot [m_{\mu\nu}]^n + 1.2[d_{\mu\nu}]^n > \mathrm{avg}(\{[m_k]^n\}_{k|g_{k,\mu\nu}=1,w_k=1});$

Where Clause Computation:
    For each $k$-th row, MPC $[w_k] = [f_{k,u} == [1]] \cdot [[o_k]^n > [100]^n]$ where $f_{k,u}$ represents $u$-th bits in the
    array $f_{k,*}$.

Group By Clause Computation (conceptual):
    For each $k$-th row, MPC $[g_{k,\mu\nu}] = [a_{k,\mu}] \otimes [s_{k,\nu}]$.

Aggregation
    For each $\mu, \nu$, MPC $[C_{\mu\nu}]^n = \sum_k \{[\mathsf{inj}(a_{k,\mu})]^n \otimes [\mathsf{inj}([s_{k,\nu}] \cdot [w_k])]^n\}$
    For each $\mu, \nu$, MPC $[S_{\mu\nu}]^n = \sum_k \{([c_k]^n \cdot [\mathsf{inj}(a_{k,\mu})]^n) \otimes ([\mathsf{inj}(s_{k,\nu})]^n \cdot [\mathsf{inj}(w_k)]^n)\}$
    For each $\mu, \nu$, MPC $[S'_{\mu\nu}]^n = \sum_k \{([m_k]^n \cdot [\mathsf{inj}(a_{k,\mu})]^n) \otimes ([\mathsf{inj}(s_{k,\nu})]^n \cdot [\mathsf{inj}(w_k)]^n)\}$
    For each $\mu, \nu$, MPC $[D'_{\mu\nu}]^n = \sum_k \{[c_k]^n \cdot [c_k]^n \cdot [\mathsf{inj}(a_{k,\mu})]^n) \otimes ([\mathsf{inj}(s_{k,\nu})]^n \cdot [\mathsf{inj}(w_k)]^n)\} - [S_{\mu\nu}]^n \cdot [S_{\mu\nu}]^n$
    For each $\mu, \nu$, MPC $[D_{\mu\nu}]^n = \mathsf{sqrt}([D'_{\mu\nu}]^n / [C_{\mu\nu}]^n)$

Having Clause Computation:
    For each $\mu, \nu$, MPC $[h_{\mu\nu}]^n = 1.3 \cdot [S'_{\mu\nu}]^n / [C_{\mu\nu}]^n + 1.2 \cdot [D_{\mu\nu}]^n / [C_{\mu\nu}]^n - [S_{\mu\nu}]^n / [C_{\mu\nu}]^n$.
    For each $\mu, \nu$, $[h'_{\mu\nu}] = ([h_{\mu\nu}]^n > 0)$.

Query Result: For each $\mu, \nu$, open all $[h'_{\mu\nu}]$. For those that are 1, open $[C_{\mu\nu}]^n$, $[S_{\mu\nu}]^n] / [C_{\mu\nu}]^n$, and $[D_{\mu\nu}]^n$.
```

Figure 10: The SQL experiment pseudocode (11:16am).

Finally, at 2:49pm on the same day, the researchers delivered a working program in SPDZ to compute the SQL query; see Figure 11.

This anecdote demonstrates the power of the SPDZ compiler and system, and is why we choose to extend this system specifically.



mpc/sql_3_.py

```
#Based on "Algorithm 173 SQL Query"

#----------------------------------------------------------------------------#
L = (20, 20, 20, 20, 20, 83, 20, 2, 10, 50)
a = 5                # 5 <= a <= 14
v = 0                # 0 <= v <= 4
N = 1000

#----------------------------------------------------------------------------#
def read_row():

#----------------------------------------------------------------------------#
def print_tup(t):
    print_ln('showing 5 integers')
    for i in range(5):
        print_ln('int %s = %s', i, t[i].reveal())

    for i in range(5, 15):
        print_ln('showing bit array %s', i)
        for j in range(L[i-5]):
            print_ln('bit %s = %s', j, t[i][j].reveal())

#----------------------------------------------------------------------------#

S = Array(L[a], sint)
C = Array(L[a], sint)

@for_range(N)
def perform_row(i):
    Ti = read_row()
    #print_tup(Ti)
    for j in range(L[a-5]):
        S[j] = S[j] + Ti[v]*Ti[a][j]
        C[j] = C[j] + Ti[a][j]

R = Array(L[a-5], sfix)
for j in range(L[a-5]):
    sfSj = sfix(0)
    sfSj.load_int(S[j])
    sfCj = sfix(0)
    sfCj.load_int(C[j])
    R[j] =sfSj/sfCj
    print_ln('R[%s] = %s; C[%s] = %s;', j, R[j].reveal(), j, C[j].reveal())

#----------------------------------------------------------------------------#
print_ln('Algorithm 173 SQL Query test done')
```

Figure 11: Working SQL code in SPDZ (4:49pm).