SECRET FIXED POINT OPERATIONS IN SHAREMIND 3

TOOMAS KRIPS, ADVISOR:JAN WILLEMSON

1. Some background

The following work aims to build secret fixed-point operations for the setting of Sharemind 3.

Fixed-point numbers is a data type where there are a fixed number of digits after the decimal point. They are in essence n-bit integers shifted to the left by a fixed number of digits. We are interested in 32-bit integers in the three-party secret-shared model—that is, a number x from $\mathbb{Z}_{32}$ is said to be secret shared among three parties if some trusted party generates x_1, x_2 from $\mathbb{Z}_{32}$ randomly, finds $x_3 = x - x_1 - x_2$ and distributes the shares x_1, x_2, x_3 to parties P_1, P_2, P_3. The Sharemind platforms are built on this. There are protocols defined that all satisfy the requirements that the parties P_1, P_2, P_3 do not learn anything about the data. The protocols are described in [1]. Examples of such protocols are protocols for adding secret values, multiplying a public value with a secret value, multiplying secret values, shifting secret values to the right by a known number of bits and others.

For some protocols, the parties occasionally need to communicate. This communication is the biggest bottleneck in our setting and thus it is important to minimize communication steps. Generally, addition and multiplication with a public constant are communication-free operations. Multiplication, division with a public constant and comparisons on the other hand need some communication. Division with a secret number is by now the operation that requires the largest amount of communication. We try to implement fixed-point numbers and some operations on them in this setting.

Three-party secret sharing has information-theoretical security:even with unlimited computational power the parties P_1, P_2, P_3 would not learn the secret. Knowing only one share, the secret can be any element of $\mathbb{Z}_{32}$ with uniform distribution. However, we will later see that we actually might wish to limit the range of our inputs or it might be possible to learn it from other sources—a persons age is probably less than $2^8 - 1$, for example.

There are essentially three fixed-point operations that are important : addition of secret values, multiplication of a public value and a secret value and multiplication of secret values. Using these operations we wish to compute more complicated functions such as e^x, $\frac{1}{1+x}$, $\sqrt{x}$ and so on. We will use different series for those functions—either Taylor series or Chebyshev series to compute the values of these functions. There is generally a tradeoff between the cost and accuracy in computing the series that depends on the number of the elements in the series.

A fixed-point number $S = s \cdot 2^{-m}$ has two important characteristics that describe the number denoted here as m and s where m is the number of digits after the decimal point and s is the number as stored in the computer memory. (Note that despite using the words "digit" and "decimal point" that refer to the decimal system, we generally use the binary system.) We also need to keep track of the data type of s but since it is generally an unsigned 32-bit integer or, sometimes, an unsigned 64-bit integer we will omit these considerations for now.

We assume that during the running-time of the program, the number of digits after the decimal point does not change.

Integers are a special case of fixed-point numbers where $m = 0$. Thus we would like that operations defined on fixed-point numbers when $m = 0$ should be the same, effect-wise and effectiveness-wise as the same operations on integers but we will later see that this is usually not possible if the integers are small.

We say that two fixed-point numbers $S_1 = s_1 \cdot 2^{-m_1}$ and $S_2 = s_2 \cdot 2^{-m_2}$ are of the same form if $m_1 = m_2$ and s_1 and s_2 have the same number of bits.

2. Basic functions

We will briefly describe our basic functions. Generally, when we describe a function, then s is a secret input and m is a public constant. We also specify beforehand how many bits before and after the decimal point we want the function to return.

- Truncate. A necessary function for the other functions. Given a secret integer s with bit expression $s_1 s_2 \dots s_k$ and a public positive integer $n \leq k$, outputs either a secret integer s' where the bitwise representation of s' is $s_1 s_2 \dots s_{k-n}$ or $s'' = s' + 1$(the latter happens preferably when $s_{k-n+1} = 1$,)—i.e. we either round downwards or upwards. For example, given the sharing of 110110101101 and $n = 4$, we want to get 11011010 or 11011010 as a result. There are two methods for this—dividing with a public power of two or converting the element to a bitwise representation and removing the necessary number of bits. Both of them are used depending on the circumstances and both are rather costly.

  One might also imagine the following approach. As we know, s is shared between the three parties as $s = s_1 + s_2 + s_3$. Each party computes $s_i \cdot 2^{-n}$, rounding up or down, as preferred in the situation. One might imagine that there might be a small error here but it would not matter as this would be very fast. However, with this approach, rather big errors can occur. Namely, imagine that $s = 0, s_1 = 2^{31}, s_2 = 2^{31}, s_3 = 0$ and we want to divide s by half. Using this approach we would get $2^{30} + 2^{30} + 0 = 2^{31}$ as an answer, which, in $\mathbb{Z}_{32}$ would be technically correct but generally, when dividing zero by two one does not wish to get a very large number as a result.

  Thus we must use either dividing with a public power of two or converting the element to a bitwise representation and removing the necessary number of bits.
- Conversion from 64-bit number to 32-bit number. This can be achieved by existing Sharemind 3 protocols. Generally we assume that the number of digits after the

decimal point in the 64-bit number is twice the number of digits after the decimal point in the 32-bit number. We need this for multiplication.

- Addition of two private fixed-point numbers is trivial and requires no communication.
- Multiplication of two secret fixed-point numbers. We want to multiply $S_1 = s_1 \cdot 2^{-m}$ and $S_2 = s_2 \cdot 2^{-m}$. We get m and s_1, s_2 as an input and wish to get s' as an output. s' should have m digits after zero.

  For multiplication, we need some extra space as when we multiply two 32 bit numbers, we might need 64 bits to store the product. Thus we first convert s_1 and s_2 to 64-bit numbers, multiply them there, truncate the result and then convert the result to the necessary format. Note that multiplication of two secret values, truncation and conversions require communication and that thus, multiplication of two secret fixed-point numbers is a fairly expensive operation.

  Note that the number of digits after zero is an important characteristic.

  Suppose we want to multiply 16 and 16, depicting them as $16 = 16 \cdot 2^0$ in both cases, that is, $m = 0$. We wish to get an output with $m' = 0$. Now, we convert both of the numbers to 64-bit numbers and multiply them. The result is $256 \cdot 2^0$. We then remove the last 32 bits. The result of this will be zero. Thus it would have been more sensible to depict 16 as $268435456 \cdot 2^{-24}$, for example. The result is $2^{56} \cdot 2^{-48}$. We remove the last 32 bits and obtain $2^{24} \cdot 2^{-16}$ which is what we wanted. Thus, if we wish to multiply them, we should store them as 2^{31} times 2^{-27}. This is made difficult by the fact that we probably do not know the values that are multiplied. Thus, generally, when depicting numbers, we should be aware of the context and what operations will be performed on them.
- Multiplication of a public fixed-point number with a private fixed-point number. This is similar to the multiplication of two secret fixed-point numbers in the manner that we need a 64-bit number to store our intermediate value. We need to use the truncating protocol, which requires communication. However, it is cheaper than multiplication of two secret fixed-point numbers as we do not require the protocol for multiplying two secret integers. On the other hand, if we want to multiply the private fixed-point number with a public integer, then this operation does not require communication which suggests that we should have two different protocols for these two cases.

3. More complicated functions

Generally, if we want to compute a more complicated function, then we do that by using some kind of a power series. Power series tend to work well in a small interval but tend to be bad approximations outside of the interval and in the edges of the interval—this is known as the Runge phenomenon and is especially egregious for the Taylor series but generally is not a very big problem when we interpolate using the Chebyshev nodes. However, we are still expected to know the approximate range of the element, despite it being secret as polynomial approximations usually only work locally.

Note that for polynomial interpolation, we only need the protocols for multiplication of secret values, multiplication of a secret value with a constant and addition. Thus, given the protocols described in the section above, we can compute any polynomial interpolations, given that we are know the approximate range of the element.

4. About our implementation

We implemented the addition protocol, the protocol of multiplying a secret value with a public constant and the multiplication of secret values in the C++ language in the Sharemind 3 framework. We implemented initialising fixed-point numbers from public floating-point numbers. We also implemented two polynomial approximations of the e^x function, one based on the Taylor series and another on the Lanczos method described in [2][p. 100-101]. The Lanczos method gives coefficients close to the Chebyshev polynomial.

We now present two tables with the errors and running times of computing the e^x function using m trailing zeroes in the fix-point representation, x as input and n as the number of elements in the approximation polynomial. We ran these tests on a computer with 2 GHz processor and 8 GB of memory. Here are the results for the Taylor approximation.

x	m	n	t	error
x=0.1	m=11	n=2	t=1.0 s	$\approx 10^{-2}$
x=0.5	m=11	n=2	t=1.0 s	$\approx 10^{-2}$
x=0.99	m=11	n=2	t=1.0 s	$\approx 10^{-1}$
x=0.1	m=11	n=4	t=2.1 s	$\approx 10^{-3}$
x=0.5	m=11	n=4	t=2.1 s	$\approx 10^{-3}$
x=0.99	m=11	n=4	t=2.1 s	$\approx 10^{-3}$
x=0.1	m=11	n=6	t=3.2s	$\approx 10^{-3}$
x=0.5	m=11	n=6	t=3.2s	$\approx 10^{-3}$
x=0.99	m=11	n=6	t=3.2s	$\approx 10^{-2}$
x=0.1	m=23	n=2	t=1.0 s	$\approx 10^{-4}$
x=0.5	m=23	n=2	t=1.0 s	$\approx 10^{-2}$
x=0.99	m=23	n=2	t=1.0 s	$\approx 10^{-1}$
x=0.1	m=23	n=4	t=2.1 s	$\approx 10^{-6}$
x=0.5	m=23	n=4	t=2.1 s	$\approx 10^{-4}$
x=0.99	m=23	n=4	t=2.1 s	$\approx 10^{-2}$
x=0.1	m=23	n=6	t=3.2s	$\approx 10^{-6}$
x=0.5	m=23	n=6	t=3.2s	$\approx 10^{-6}$
x=0.99	m=23	n=6	t=3.2s	$\approx 10^{-4}$

We see that if the number of digits after the decimal point is small, then using an approximation polynomial with more elements does not give us any additional accuracy—indeed, when we have only 11 binary digits after the decimal point then the smallest value that we can use is 2^{-11} which is about 0.0005. Thus we see that the number of digits after the decimal point should be as big as possible.

Here are the results for the Chebyshev approximation.

x	m	n	t	error
x=0.1	m=11	n=2	t=1.0 s	$\approx 10^{-2}$
x=0.5	m=11	n=2	t=1.0 s	$\approx 10^{-2}$
x=0.99	m=11	n=2	t=1.0 s	$\approx 10^{-1}$
x=0.1	m=11	n=4	t=2.1 s	$\approx 10^{-3}$
x=0.5	m=11	n=4	t=2.1 s	$\approx 10^{-3}$
x=0.99	m=11	n=4	t=2.1 s	$\approx 10^{-2}$
x=0.1	m=23	n=2	t=1.0 s	$\approx 10^{-2}$
x=0.5	m=23	n=2	t=1.0 s	$\approx 10^{-2}$
x=0.99	m=23	n=2	t=1.0 s	$\approx 10^{-1}$
x=0.1	m=23	n=4	t=2.1 s	$\approx 10^{-6}$
x=0.5	m=23	n=4	t=2.1 s	$\approx 10^{-5}$
x=0.99	m=23	n=4	t=2.1 s	$\approx 10^{-4}$

We see that occasionally Chebyshev outperforms the Taylor approximation in points that are not close to 0. Note that for a fixed m and n in a given method, the running times do not vary as if it did, running time would leak information.

5. Possible uses

Fixed-point numbers are useful if we know the number of digits after the decimal point or the approximate range where the number belongs to as they are probably faster than floating-point numbers. For example, sums of money generally have two digits after the decimal points in decimal for which we could use binary numbers that have eight digits after the decimal point. Another possible use can be probabilities -we know they are between 0 and 1 and thus it makes sense that all the digits are after the decimal point. We generally assume that the user knows the number of digits after the decimal points that is wise to use.

However, if the range of the numbers varies much then it is better to use floating-point numbers that are also supported by Sharemind 3.

References

[1] *High-performance secure multi-party computation for data mining applications* D.Bogdanov, M.Niitsoo, T.Toft, J.Willemson, International Journal of Information Security November 2012, Volume 11, Issue 6, pp 403-418

[2] *Arvutusmeetodite käsiraamat*, M.Levin, S.Ulm, Kirjastus Valgus, Tallinn, 1966