A Domain-Specific Language for Low-Level Secure Multiparty Computation Protocols

Peeter Laud
Cybernetica AS
peeter.laud@cyber.ee

Jaak Randmets
Cybernetica AS
University of Tartu
jaak.randmets@cyber.ee

ABSTRACT

SHAREMIND is an efficient framework for secure multiparty computations (SMC). Its efficiency is in part achieved through a large set of primitive, optimized SMC protocols that it makes available to applications built on its top. The size of this set has brought with it an issue not present in frameworks with a small number of supported operations: the set of protocols must be maintained, as new protocols are still added to it and possible optimizations for a particular sub-protocol should be propagated into larger protocols working with data of different types.

To ease the maintenance of existing and implementation of new protocols, we have devised a domain-specific language (DSL) and its optimizing compiler for specifying protocols for secure computation. In this paper, we give the rationale of the design, describe the translation steps, the location of the compiler in the whole SHAREMIND protocol stack, and the results obtained with this system.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Cryptographic controls*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

Keywords

Secure multi-party computation; secure computation outsourcing; additive secret sharing; compiler; protocol optimization

1. INTRODUCTION

Secure multiparty computation (SMC) has become more and more practical in recent years, with the appearance of several SMC frameworks [2, 14, 7, 10, 20, 34, 43, 33] and noteworthy applications [9, 25, 3]. Existing SMC frameworks use different protocol sets for achieving privacy. Several frameworks implement the *arithmetic black box* (ABB) [15], the methods of which are called during the runtime of a privacy-preserving computation by the SMC engine in the order determined by the specification of the computation. An ABB implementation consists of a data representation for private values, somehow sharing them among the computing parties, and a set of protocols for operations on shared values. When executing a protocol from this set, the computing parties provide the shares of the operands as inputs to that protocol, and receive the shares of the result as outputs. An ABB must at least contain the methods for linear combination and multiplication of private integers (in order to be Turing-complete), but it contains more in typical implementations. For yielding secure applications making use of SMC, the ABB implementation must be universally composable [11]. In this case, the protocols of the ABB can be invoked in any order, sequentially or in parallel, without losing security guarantees.

SHAREMIND SMC framework [7] features an exceptionally large ABB. Besides the operations listed above, it also contains comparisons, bit extraction, bit conversions, division of arbitrary-width integers [8], as well as a full set of floating-point [23] and fixed-point [12] operations, including the implementations of elementary functions. More often than not SHAREMIND protocols are specified in a compositional style forming a hierarchy, with more complex protocols invoking simpler ones. For example, floating-point operations typically use fixed-point operations which in turn use integer operations [28]. The choice to expand the ABB of SHAREMIND has been validated by the multitude of privacy-preserving applications it has been used for, including genome-wide association studies [22], prediction of satellite collisions [23], and a privacy-preserving statistical analysis tool [4].

The implementation of protocols for ABB operations is an error-prone and repetitive task. Manual attempts to optimize complex protocols over the composition boundaries is a laborious task, prone to introduce errors and make the library of protocols unmaintainable. Implementation is made more difficult due to the fact that protocols need to work for various different integer widths and many of the abstractions in the implementation language (such as virtual function calls in C++/Java) entail unacceptable run-time overhead. The task of building and maintaining implementations of protocols is naturally answered by introducing a domain-specific language (DSL) for specifying them.

The DSL allows us to specify the protocols in a manner similar to their write-up in papers on SMC protocols. This specification is compiled and linked with the SHAREMIND platform. A different language [6] is used for specifying the privacy-preserving applications as a composition of these protocols. Having different languages for implementing different levels of the privacy-preserving computation allows us to apply optimizations most suitable for each level, and improves the user experience by allowing us to tailor the languages for the specific domain. Protocols are specified and implemented in a declarative style, but applications built on top of the primitive protocols are specified in an imperative style as a sequence of protocol invocations.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813664 .

In this paper we describe our protocol DSL. We discuss its details and their rationale, show how it enables us to easily build protocols for complex operations, and describe our experience in using it, both in terms of performance and maintainability. In Sec. 2 we introduce our language through a number of examples, allowing us to demonstrate the impouurtant details of the DSL. Sec. 3 discusses how the language is defined, compiled, optimized and located in the Sharemind framework. In Sec. 4 we discuss the security of the protocols implemented in our DSL, and how the DSL toolchain supports the development of secure protocols. In Sec. 5 we show how the use of the DSL has improved the performance of the protocols employed in Sharemind. In Sec. 6 we discuss related developments and in Sec. 7 give our conclusions.

We note that other SMC frameworks with large ABBs have been proposed [43, 24]. Also, there exist SMC applications that have been implemented essentially on top of an ABB with a large number of SMC protocols [25, 17]. All these could potentially benefit from the DSL we are proposing and the architecture it implies.

2. OVERVIEW OF THE PROTOCOL DSL

We sought to create a programming language that facilitates the implementation of SMC protocols in similar style to their specification in [7] and [23]. Our experience in implementing SMC protocols in C++ for Sharemind showed that the aspect most hindering our productivity and performance was the lack of composability. Namely, our C++ framework is designed to allow a single protocol to be executed on many inputs in parallel (in SIMD style) but the parallel execution of two different protocols has only been realizable by interleaving both protocols and manually packing all network messages together. This effort was very time-consuming even for medium sized protocols. The lack of composability has led to poor readability, maintainability and modifiability. Fixing a bug means making changes in every place the modified protocol has been copied to. This has often led to sacrificing performance for readability and development time.

Hence the key design principle of the protocol DSL is to always put composability first: whenever the data-flow dependencies allow, the protocols are executed in parallel, no matter the order they occur in the code. For performance reasons, it is very important to keep the round complexity of protocols low, as a network roundtrip is orders of magnitude slower than the time it takes to evaluate the (arithmetic) gates in the protocols. In implementing protocols in C++, large parts of the code dealt with how the network messages are packed and how they are sent to other computing parties – we wanted to automate this process. In the protocol language we have simplified this aspect and the programmer only has to specify what values are used by other parties, but not how or when they get there. The compiler minimizes the round count, chooses how values are packed into messages and deals with sending and receiving the network messages automatically.

We also wanted to have an optimizing compiler for the language so that the programmer does not have to optimize manually when compositions introduce possibilities for it (e.g. by partially applying a protocol to a constant value). The protocol DSL is a functional language enabling a declarative style and letting the programmer manipulate protocols in a higher-order manner. For instance, it is natural to apply a protocol to each element of a vector by using a higher-order `map` operator.

The language supports type level integers (called size types) and arithmetic on them. Functions polymorphic in the number of input bits allow the protocols to be specified once for all input lengths. The language is mainly designed for implementation of additive secret sharing schemes but is not strictly limited to that.

```
def reshare : uint[n] -> uint[n] = \u ->
  let r = rng ()
  in u + r - (r from Next)
```

Listing 1: Resharing protocol

2.1 Additive secret sharing

In Sharemind SMC framework the private values are additively shared among the parties performing the secure computation. The sharing is over a finite ring. Different rings of the form $\mathbb{Z}_{2^n}$ and $\mathbb{Z}_2^n$ can be used. The protocols of the framework perform operations with secret-shared values, receiving the shares of the operands as inputs and delivering the shares of the results.

Even though our protocol DSL can support an arbitrary number of parties, the protocols of Sharemind have been designed specifically for three parties and in this paper we also focus on this case. A secret-shared value $x \in R$, where R is one of the rings $\mathbb{Z}_{2^n}$ or $\mathbb{Z}_2^n$, is represented as a triple (x_1, x_2, x_3) with the i-th party holding $x_i \in R$, satisfying $x_1 + x_2 + x_3 = x$, where the addition is in ring R. A nice property of additive scheme is that integers can be added with no network communication by adding the respective shares. Most other operations require at least one communication round.

In Sec. 4 we provide more detailed discussion of the security of SMC protocols in general and Sharemind's protocols in particular, as well as the support offered by the protocol DSL and the Sharemind framework for ensuring the security of protocols. Briefly, our protocols provide security against a single honest-but-curious party, meaning that for any single computing party, its view could be generated from its inputs only, and does not depend on the inputs and outputs of the other parties.

To ensure such independence, *resharing* is a commonly used in implementing Sharemind's set of protocols. The significance of the resharing protocol is in providing fresh shares for input value so that the new shares can be used as inputs for further protocols without giving any information about the input shares. The implementation of the resharing protocol [8, Alg. 1] in our DSL is depicted in Listing 1. As the input, the i-th party receives the share u_i of some private value $u \in \mathbb{Z}_{2^n}$. In order to obtain the output share u_i', the i-th party generates a random value r_i, sends it to the previous party and finally adds the difference of generated and received random value to the input. All arithmetic is performed modulo 2^n. The final shares obtained in the protocol will be $(u_1', u_2', u_3') = (u_1 + r_1 - r_2, u_2 + r_2 - r_3, u_3 + r_3 - r_1)$. We see that $u_1' + u_2' + u_3' = u_1 + u_2 + u_3 = u$, i.e. the output value is the same as the input value but u_i' is independent of u_i.

In the protocol DSL implementation Listing 1 we define a variable `reshare` of type `uint[n] -> uint[n]` denoting a function taking an n-bit unsigned integer to an n-bit unsigned integer (the number of bits can be arbitrary). After the equals sign the variable definition follows: in this case it is defined to be a function taking an argument called `u`. The body of the function follows after the arrow and it first randomly generates a variable `r` and then evaluates to expression `u + r - (r from Next)`. It is important to note that we have not explicitly stated which computing party performs which computation. The default mode of operation in the protocol DSL is that every computing party executes the same code. For instance, each party generates a random value independently even though it is only written once in the code. The types of variables are derived via type inference: from the type of the function we know that the input variable `u` must be an n-bit integer, and because addition op-

```
def reshareToTwo : uint[n] -> uint[n] = \u ->
  let {1}
    r2 = rng ()
    r3 = u - r2
  in case:
    1 -> 0
    2 -> u + (r2 from 1)
    3 -> u + (r3 from 1)
```

Listing 2: Protocol for resharing between two parties

erates on integers of same bit width we know that the randomly generated variable `r` must also be an n-bit integer.

In Listing 1, we see how the transmission of values between parties is specified. The receiver of the value states which value and from which party it wishes to receive. The **from**-construct is a somewhat unusual networking primitive but it is a good fit for us as it does not specify when or in what order are network messages delivered. It is up to the code generator to select how to pack values into network messages and when to send them. While it is definitely true that a programmer can easily construct more efficient networking schemes for small protocols, our experience shows that doing so for medium sized and large protocols is an extremely time-consuming and labor intensive task. The Sharemind's architecture allows us to select between hand-tuned and generated protocols.

Sometimes it is useful to reshare a value in such a way that one of the parties holds 0 as its share. Such resharing protocol is given in Listing 2. In this protocol, given the shares of an input $u = (u_1, u_2, u_3)$, the first computing party generates a random value r_2, sends it to the second computing party and sends $r_3 = u_1 - r_2$ to the third computing party. The second and third computing parties add the received values to their input shares. The resulting shares are $(0, u_2 + r_2, u_3 + (u_1 - r_2))$ which sum to the original value u.

The protocol in Listing 2 demonstrates various features of the language: the ability to perform computation and to define values for only some subset of parties, the ability to branch the computation depending on the evaluating party and finally the ability to receive values from certain fixed parties. The variables `r2` and `r3` are only defined by the first computing party. The result of the function body is computed differently depending on the computing party: the first computing party always returns 0, while the second [resp. third] computing party adds `r2` [resp. `r3`] received from the first party to its input share.

2.2 Multiplication protocol

The algorithm for multiplying two additively shared numbers $u, v \in \mathbb{Z}_{2^n}$ is based on a simple equality given by distributivity of multiplication over addition: $(u_1+u_2+u_3)(v_1+v_2+v_3) = \sum_{i,j=1,1}^{3,3} u_i v_j = \sum_{i=1}^{3}(u_i v_i + u_i v_{p(i)} + u_{p(i)} v_i)$, where $p(i)$ denotes the previous index (p maps $2 \mapsto 1$, $3 \mapsto 2$ and $1 \mapsto 3$). This equation is directly mapped to code in Listing 3 by letting the i-th party compute the term $w_i = u_i v_i + u_i v_{p(i)} + u_{p(i)} v_i$. To achieve security and privacy the algorithm reshares both of the inputs and the output. Notice the let-expression overshadowing input variables `u` and `v` with same names. We can see the similarity to Algorithm 2 in [8] but again, the presentation is much more concise.

Textually the resharing call on the output occurs after the rest of the code but notice that there are no data dependencies that forbid us from performing the network communication of all the reshare calls in parallel during the first communication round. This is exactly what happens in practice where we try to minimize the number of communication rounds required by the protocol. The

```
def mult : uint[n] -> uint[n] -> uint[n] = \u v ->
  let
    u = reshare u
    v = reshare v
    w = u * v + u * (v from Prev) + (u from Prev) * v
  in reshare w
```

Listing 3: Multiplication protocol

simplest approach to achieve this is to greedily send values with messages in the earliest communication round possible. We will see in Sec. 3.5 that during automatic optimizations the round count of multiplication protocol is reduced to one.

Usually we specify the protocols for integers of arbitrary bit-width n, such as the multiplication protocol here, but in the concrete system the protocol implementations are instantiated to bit-widths that computers support natively. In most cases protocols are specialized to operate on 8-, 16-, 32- and 64-bit integers but in general the bit-widths we specialized the protocols to are not restricted. The only limitation is that we do not allow the choice of bit-width happen dynamically; it always has to be fixed before executing the code, during compilation of protocols. Support for arbitrarily large integers has turned out to be extremely useful. For example, integer division protocol internally uses larger than 128-bit integers. In addition to that, as we will later see, fixed-point computations (that are used to implement floating-point operations as in [28]) can be sped up by starting operations on large integers and gradually cutting back during the protocol.

2.3 Bit-level protocols

Many of the high-level protocols in Sharemind are implemented in terms of bit-level operations. Accessing bits of additively shared values is a non-trivial task. For example, to extract the highest (most significant) bit of a 2-bit number we not only have to consider the highest bits of the shares but also have to take into account the possibility that the sum of the lowest bits of the shares may overflow and influence the value of the highest bit.

Consider the special case of additive secret sharing over the ring $\mathbb{Z}_2$. The multiplication over $\mathbb{Z}_2$ acts as boolean conjunction $\wedge$ and the addition operation as the XOR operation $\oplus$. We can extend this to bitwise additive secret sharing over the ring $\mathbb{Z}_2^n$ where the bitwise XOR of two private values can be computed as the bitwise XOR of respective shares and the bitwise conjunction can be computed with a protocol similar to Listing 3. Bitwise negation is computed by an odd number of parties negating their shares, and disjunction is computed using conjunction and negation via De Morgan's laws. We call this kind of bitwise additive secret sharing *XOR sharing*.

Prefix-or is a primitive bit-level protocol that is often used inside higher-level operations. The prefix-or of a value $\vec{u} \in \mathbb{Z}_2^n$ is obtained by propagating its most significant 1 bit downwards. For example, the prefix-or of the 8-bit number 00101100_2 is 00111111_2. If u_i is the i-th bit of $\vec{u}$, then the prefix-or of $\vec{u}$ is $\vec{v}$, where $v_i = \bigvee_{j=i}^{n} \vec{u}_i$.

The implementation of prefix-or is shown in Listing 4. We use the `disjBit` protocol for computing disjunction of a XOR shared number and a bit, and a built-in function `++` for array concatenation. Our DSL allows to manipulate integers as bit-arrays where lower indices denote less significant bits. The `prefixOR` function is defined recursively: we split the input into two roughly equal parts (`u[m .. n]` denotes the slice of the array `u` from index m until the index $n-1$) and recursively compute the prefix-or of the parts. We concatenate the resulting parts, but if the lowest bit of the higher half is set, then every bit of the lower half of the result must be set

```
def disjBit : uint[n] -> bit -> uint[n]
def ++ : arr[a,n] -> arr[a,m] -> arr[a,n+m]
def prefixOR : uint[n] -> uint[n] = \u ->
  if (n < 2) then u else
  let
    x = prefixOR u[0 .. n/2]
    y = prefixOR u[n/2 .. n]
    x = disjBit x y[0]
  in x ++ y
```

Listing 4: Prefix-or protocol

```
def addXor : uint[n] -> uint[n] -> uint[n]
def bitextr : uint[n] -> uint[n] = \u ->
  let
    v = reshareToTwo u
    x = case: 2 -> v | {1,3} -> 0
    y = case: 3 -> v | {1,2} -> 0
  in addXor x y
```

Listing 5: Bit extraction

as well. Prefix-or of a 0- or 1-bit number is the number itself; this is the recursion base.

The implementation of prefix-or in Listing 4 demonstrates recursion over type level naturals: the if-check is over a type predicate $n < 2$ and recursive calls are performed on $n/2$-bit and $(n-n/2)$-bit numbers. The algorithm terminates as $n/2$ and $n - n/2$ are strictly smaller than n whenever $n \geq 2$. The protocols we have seen thus far take a constant number of rounds regardless of the number of input bits and the prefix-or is the first exception, taking $O(\log n)$ communication rounds for an n-bit input.

The bit level protocols are used as building blocks for high-level additive protocols. We may need to convert between additive sharing and XOR sharing. Converting a bit $u \in \mathbb{Z}_2$ to an additively shared integer in $\mathbb{Z}_{2^n}$ is not completely straightforward, because the shares of u may also sum up to 2 or 3 in $\mathbb{Z}_{2^n}$. A more elaborate protocol `shareconv` is required for converting a bit to an additively shared integer: we do not give its listing here but it can easily be adapted from [8] with the tools we already have. Notice that using the share conversion protocol we can convert an arbitrary bit-width XOR shared integer $\vec{u} \in \mathbb{Z}_2^n$ to an additively shared one by converting each bit $\vec{u}_i$ to $v_i \in \mathbb{Z}_{2^n}$ and then computing the dot product with successive powers of two: $\sum_{i=0}^{n-1} 2^i v_i$.

We have seen how to convert XOR shared data to additively shared data but the converse is also required. The protocol for doing that (Listing 5) is conceptually very simple. Given an additively shared value $v \in \mathbb{Z}_{2^n}$ we first reshare it between second and third party as $(0, v_2, v_3)$ such that $v = v_2 + v_3$. The reshared value can then be viewed as two XOR-shared values $x = (0, v_2, 0)$ and $y = (0, 0, v_3)$. To compute the XOR-shared sum (simply adding them in $\mathbb{Z}_{2^n}$ will produce an additively shared sum) of x and y we implement an adder circuit. All the necessary tools for doing so – elementary bitwise operations and recursion – are provided. We have omitted the implementation of `addXor` function as it is relatively straightforward but verbose.

In certain protocols we need to compute the XOR-shared *characteristic vector* $\vec{c} \in \mathbb{Z}_2^m$ of an additively shared $u \in \mathbb{Z}_{2^n}$ (represented as triple (u_1, u_2, u_3)), for which we know that it is in some range $0 \leq u < m = 2^k$, where m is relatively small. By definition, characteristic vector satisfies $c_i = 1 \Leftrightarrow i = u$. The protocol for this purpose is given in Listing 6. It first rotates $1 \in \mathbb{Z}_2^m$ left (using the

```
def rol : m >= n => uint[m] -> uint[n] -> uint[m]
def chVector : m >= n > 0 => uint[n] -> uint[m] = \u ->
  let
    u = reshare u
    c = case:
      1 -> rol (1, u)
      {2, 3} -> 0
    c = xorReshareToTwo c
  in case:
    1 -> 0
    {2, 3} -> rol (c, u from 2 + u from 3)
```

Listing 6: Characteristic vector

```
def countUp : uint[n] -> arr[uint[n],m]
def conjBit : uint[n] -> bit -> uint[n]
def shiftr : n >= m > 0 => uint[n] -> uint[m] -> uint[n] =
  \u s -> let
    u = bitextr u
    v = map (\i -> u >> i) (countUp 0)
    bs : uint[n] = chVector s
    rs = zipWith conjBit v bs
    r = foldl (\x y -> x ^ y) 0 rs
  in xorToAdditive r
```

Listing 7: Private shift right

function `rol`) by u_1 and XOR shares the resulting value between parties 2 and 3. The receiving parties sum the input shares u_2 and u_3 and rotate the received values by the sum to produce the result. As a result we have rotated $1 \in \mathbb{Z}_2^m$ by u positions (first party rotating it by u_1 and the rest rotating it by $u_2 + u_3$). Because the shares of u can be larger than m the rotations are performed modulo m. This protocol takes constant number of communication rounds regardless of the number of input or output bits. It could be converted to a bit extraction protocol (the one in Listing 5 takes $O(\log n)$ rounds) using no further communication, but its network communication scales linearly with the bound m.

2.4 High-level protocols

One of the simplest high-level protocols that Sharemind implements is a bit-shift right on additively shared data (Listing 7). This protocol is often used to implement floating-point operations but is also useful as a stand-alone protocol. To shift an additively shared $u \in \mathbb{Z}_{2^n}$ right by an additively shared $s \in \mathbb{Z}_{2^m}$ we first convert u to XOR shared integer $\vec{u}$. The XOR shared representation allows us to calculate all possible shifts $v_i = \vec{u} \gg i$ by public values $0 \leq i < m$. As the result we pick the correct one out of all the possible shifts by computing r as $\bigoplus_{i=0}^{n-1} v_i \wedge (s = i)$. The protocol `conjBit` is applied pointwise to elements of `v` and `b` using higher-order function **`zipWith`** and all of the invocations of the argument protocol are executed in parallel. Listing 7 also demonstrates the parallel composition of previously defined non-trivial protocols: `bitextr` and `chVector`. The meaning of higher-order functions **`map`**, **`zipWith`** and **`foldl`** is standard (see e.g. [32]).

We have used the protocol DSL to implement floating-point arithmetic and most of the primitive operations from [23]. Briefly, in Sharemind a floating-point number N is composed of three parts: sign bit s, significand f, and exponent e such that $N = (-1)^s \cdot f \cdot 2^e$ where the significand is always in the range $1/2 \leq f < 1$. In the protocol DSL we represent the sign bit, exponent and single-precision fractional part with 1-, m- and n-bit unsigned integers

respectively (for single- [resp. double-]precision floats we have $m = 16$ and $n = 32$ [resp. $n = 64$]).

The fractional part f is represented as an unsigned integer where the most significant bit denotes $1/2$, the second highest one $1/4$ and so on. This lets us approximate real values in the range $[1/2, 1)$. We require that the representation is normalized, meaning that the most significant bit is always 1. The only exception to that is when we want to represent zero, in which case $f = 0$. Unlike IEEE floating-point numbers we explicitly store the highest bit.

As a final example we demonstrate a protocol for computing the inverse of a floating-point number. The intuitive idea is that for a floating-point number $N = (-1)^s \cdot f \cdot 2^e$ we have

$$\frac{1}{N} = (-1)^s \frac{1}{f2^e} = (-1)^s \frac{1}{2f} \frac{1}{2^{e-1}} = (-1)^s \cdot \frac{1}{2f} \cdot 2^{1-e} \ .$$

This already gives us a suitable floating-point representation for every $f \in (1/2, 1)$ because in such case $\frac{1}{2f} \in (1/2, 1)$. The case when the fractional part of the input is $1/2$ or close to it is self-correcting as our algorithm rounds the result down and prevents it from overflowing. This gives us the recipe for computing the inverse of a floating-point number. We first compute $1 - f$ and interpret it as a fixed-point number with a single binary digit before the radix point. To do that we divide $-f$ by 2 (negation and division computed for an unsigned integer). A fixed-point format with a single binary digit before the radix point allows us to represent values in the range $[0, 2)$ and the extra digit is needed because inverse yields us a value in the range $[1, 2)$.

By setting $x = 1 - f$ we can compute $1/f$ using the equality $1/f = 1/(1 - x) = \sum_{i=0}^{\infty} x^i = \prod_{i=0}^{\infty} \left(x^{2^i} + 1\right)$. Evaluating just the first k terms of the product gives the maximum error of about 2^{-2^k} at $x = 1/2$. This means that for a single-precision floating-point number it is sufficient to only compute the first 5 terms. To get the fractional part of the result all that is left to do is to evaluate that expression on fixed-point numbers. Note that this approximates $1/f$ as a fixed-point number with one digit before the radix point, but reinterpreting that as a fixed-point number with no digits before the radix point yields the approximation for $1/(2f)$.

To find $1/f$ we need to compute powers of $x = 1 - f$ and multiply the terms incremented by one to approximate the wanted value. Therefore, fixed-point multiplication is needed. Let u and v be $(1 + n)$-bit fixed-point numbers with a single digit before the radix. To compute the product $u * v$ (assuming that the result does not overflow) we extend both of the numbers to $1 + 2n$ bits, multiply them and then cut away n least significant digits of the result. This is a rather expensive operation: to extend the numbers we need to compute their overflow bits and to cut away least significant digits we again need to check if those digits overflow. The overflow bits have to be computed because extending a number to a larger one has the same problem that we already faced with extending a single bit integer to a larger one. One of the ways to compute the overflow bit is to tailor the bit extraction protocol for this purpose (a more efficient method is provided in [8]).

If we know, ahead of time, that we are performing some multiplications in a row, for example, when computing a product of several numbers, we can optimize the computation by eliminating the need to extend the numbers before every multiplication. If we know that we are performing exactly r multiplications on u we can instead immediately extend it to $1 + (1 + r)n$ bits and on every successive multiplication remove the lowest n bits.

The implementation for floating-point inverse is presented in Listing 8. We have used but not defined various helper functions: a) `publicShiftr` for shifting an additively shared value right by some public value; b) `choice` for obliviously choosing between two

```
type float[n,m] = uint[1] * uint[n] * uint[m]
def bias : unit -> uint[m]

def floatInv : n > m > 3 => float[n,m] -> float[n,m] =
  \N -> let
    (s, f, e) = N
    x = publicShiftr (-f) 1 // x = 1 - f
    f' = fixInv x
    e' = share((bias() + 1) << 1) - e // 2 - e
    // b checks if 1/(2f) < 1/2
    b = trunc (publicShiftr f' `(n - 1))
    half = share (1 << `(n - 1))
    f' = choice b f' half // correct fraction
    e' = e' - shareconv b // correct exponent
  in (s, f', e')

def fixInv : n > 0 => uint[n] -> uint[n] = \x ->
  let
    x : uint[n + 5*(n - 1)] = extend x
    one = share (1 << `(n - 1))
  in fixInvLoop (x[0 .. n + 4*(n - 1)] + one) x

def fixInvLoop : n > 0 =>
    uint[n+(r+1)*(n-1)] ->
    uint[n+(r+2)*(n-1)] -> uint[n] =
  \acc xPow -> let
    xPow : uint[n+(r+1)*(n-1)] = cut (square xPow)
    one = 1 << `(n - 1)
    acc = cut (mult acc (xPow + one))
  in if (r == 0) acc else fixInv acc xPow
```

Listing 8: Floating-point reciprocal protocol

additively shared integers; c) `share` for sharing a public value by having two of the parties pick 0 as their shares; d) `cut` for cutting away some least significant bits of an additively shared integer; and e) `extend` for converting an additively shared integer to a larger one. Floating-point numbers are represented by a triple consisting of a 1-bit sign, n-bit fractional part and m-bit exponent. The type synonym `float[n,m]` is provided for this. The function `bias` returns the bias for an m-bit exponent (we omit the definition), the function `fixInv` computes the inverse of an n-bit fixed-point number with a single digit before the radix point and finally `floatInv` computes the inverse of a floating point number.

If the input had a fractional part very close to 1 then $1/(2f)$ is very close to $1/2$. During the computation this may be rounded down and the highest bit can become 0 resulting in a denormalized float. To avoid this, we need to check the highest bit of the to-be fractional part – we denote it with `b`. This is computed by shifting the fractional part right by $n-1$ bits. If the highest bit turns out to be 0 then we know that the input had a fractional part very close to 1 and the result was rounded down too much during the computation. In this case we correct both the resulting fraction and the exponent.

Initially we implemented the reciprocal protocol using fixed-point polynomial evaluation technique as in [23]. However, the protocol DSL enabled us to rapidly try out different implementations and optimizations and we quickly found out that the approach presented here is superior to polynomial evaluation both in speed and in precision. Implementing the protocol in optimized manner in our C++ framework would have been a major undertaking.

3. THE CORE PROTOCOL LANGUAGE

In this section we will formalize the core of the protocol DSL. The code examples presented previously do not match the syntax provided here perfectly but can be translated to the core language

Size literals	c	$\in$	$\mathbb{N}_0$
Party nr.	p	$\in$	$\mathbb{N}_1$
Sources	q	::=	$p \mid \texttt{Prev} \mid \texttt{Next}$
Expressions	e	::=	$x \mid \lambda x.\ e \mid \Lambda\alpha.\ e$
		$\mid$	$e_1\ e_2 \mid e\ \tau \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$
		$\mid$	$\texttt{if}\ C\ \texttt{then}\ e_1\ \texttt{else}\ e_2$
		$\mid$	$\texttt{case:}\ e_1, \ldots, e_k \mid e\ \texttt{from}\ q$
Programs	M	::=	$\epsilon \mid \texttt{def}\ x : \sigma\ =\ e\ M$
Constraints	C	::=	$\varepsilon \mid C_1 \wedge C_2 \mid s_1 \sim s_2 \mid s_1 < s_2$
Monotypes	τ, s	::=	$\alpha \mid \texttt{unit} \mid \texttt{bit} \mid \texttt{arr[}\tau\texttt{,}\ s\texttt{]} \mid \tau_1 \to \tau_2$
Size types		$\mid$	$c \mid s_1 + s_2 \mid s_1 * s_2 \mid s_1/s_2$
Polytypes	σ	::=	$\forall\overline{\alpha}.C \Rightarrow \tau$

Figure 1: Syntax of the core protocol DSL

with relatively little effort. For some constructs we have provided syntactic rewriting rules. A major difference is that the core language does not infer type parameters automatically and expects explicit type applications. In the compiler implementation the type arguments are inferred whenever possible during type checking.

The syntax of the language is presented in Fig. 1. Expressions e of the language include the standard constructs for lambda calculus: variables, function applications, lambda-abstractions and let-expressions. In addition to that the language includes conditional expressions over size predicates, case-expressions for branching depending on the computing party, and from-expressions for performing network communication. An expression e is always evaluated by a set of computing parties that may communicate between each other. By default, all the parties evaluate the same expression, but every computing party does not always hold a result for the given expression. We will see that case-expressions allow the computation to branch depending on the evaluating party but may also omit the value for some parties. The let-expressions are similar to those in ML. They are not recursive and shadow (override) previous variable definitions with the same name.

The from-expression e `from` q is used for network communication and states that the current evaluating party gets the value of e from source q. For example, for the second computing party the expression x `from` 1 evaluates to the first computing party's value of the variable x. The source q is not only restricted to concrete parties but may also denote the next or the previous computing party.

The conditional expression `if` C `then` e_1 `else` e_2 evaluates to e_1 if type constraint C holds and otherwise evaluates the expression e_2. When type checking a branch the fact that C does or does not hold may be used depending on the branch. The protocol DSL compiles to an intermediate representation (IR) with no branching constructs, meaning that the source code may only contain loops that are statically bounded. The mixture of supporting type-level integers and providing the ability to branch over them facilitates writing recursive code and cleanly segregates values that might only be dynamically known (regular values) from values that are definitely statically known (types). Recall that Sharemind protocols are always instantiated to concrete bit-widths.

The case-expression `case:` $e_1, \ldots, e_k$ evaluates to e_i for the i-th party where k is the number of computing parties. This construct looks quite different to what we have seen in the examples above but high-level code can be straightforwardly trasnalted to this form. If a case-expression has any uncovered cases we can add them by mapping to undefined values. As an example, a case expression with no branches has no value for any of the computing parties (it is undefined everywhere). A set of parties can be implemented by binding the expression to a fresh variable, replacing the expressing with the variable and duplicating that branch for each party.

A program M of the language consists of a sequence of variable definitions. All top-level bindings must be annotated with types and the definitions may be mutually recursive unlike the regular let-expressions.

3.1 Type system

The type system of the language is inspired by Cryptol [31]. We have opted for strict and static type checking with type inference: a classic Hindley-Milner type system [13] extended with type constraints (predicates) over type-level natural numbers. A regular type τ is either a variable α, the `unit` type having only a single value, the `bit` type having two values 0 and 1, an array `arr[`τ_1`,` s`]` of length s containing elements of type τ_1, or a function $\tau_1 \to \tau_2$ taking arguments of type τ_1 and returning values of type τ_2. Because most protocols operate on integer values we use `uint[`n`]` as a synonym for an array of n bits for the sake of conciseness and readability. The language additionally supports n-ary tuples and data structures, but we have omitted them here as they are a relatively straightforward addition to the language. The protocol DSL has two different kinds of types: regular data types, and size types for denoting lengths of arrays. A size type s is either a variable n, a natural number $c \in \mathbb{N}_0$, or an arithmetic expression of size types.

The type system is simply an instantiation of OutsideIn(X) where X has been chosen to be integer constraints. The general type checking algorithm is described in [41]. The type checking of the protocol DSL is made easier by our requirement to annotate all polymorphic types: top-level bindings must be annotated and no generalization is performed when type checking let-expressions. Note that the type system of the language is definitely not complete: in order to type check arbitrary programs we need to be able to solve arbitrary (non-linear) systems of equations. In practise this has not turned out to be a hindrance as almost all of the constraints are very simple and easily dispatched by Z3 [16] SMT solver.

3.2 Semantics

The semantics of the language is relatively straightforward by exploiting the fact that programs of the language must always terminate. There are two kinds of values in the protocol DSL: functional values (either value or type abstraction), and tuples of primitive values where the i-th component denotes the value that the i-th party has. When we say that some value is undefined everywhere we mean that it is a tuple consisting of bottom values $\bot$.

The semantics is in small-step style. The transition rules are either from expression to another $e \xrightarrow{p} e'$ or from expression to a value $e \xrightarrow{p} v$. All of the transitions are annotated with probabilities (omitted if equal to 1). The meaning of constraints $[\![C]\!] \in \{0, 1\}$ is defined in the obvious manner.

The evaluation rules for the three party case are given in Fig. 2. Mostly they are straightforward lambda-calculus rules: we evaluate expressions under the evaluation context C and substitute variables in case of function application and let-expression. Evaluation is performed strictly except for lambda (or type) abstractions and if-expressions. The from-expression rearranges the components of the tuple (in syntax we have $i \equiv \langle i, i, i\rangle$, `Next` $\equiv \langle 2, 3, 1\rangle$ and `Prev` $\equiv \langle 3, 1, 2\rangle$). For case-expression all subexpressions are evaluated and then correct components are picked out of the branches. The generation of a random bit by three parties chooses each possible set of three bits with equal probability.

Implementing this semantics would result in an extremely inefficient evaluator. It would constantly compute values that will never be used (due to the case-construct dropping them) and often prop-

$$\frac{e \xrightarrow{p} e'}{C[e] \xrightarrow{p} C[e']}$$

$$\frac{}{(\lambda x.\, e)v \to e[x \mapsto v]} \qquad \frac{}{(\Lambda \alpha.\, e)\tau \to e[\alpha \mapsto \tau]}$$

$$\frac{}{\{v_1, v_2, v_3\}\ \texttt{from}\ \langle i, j, k\rangle \to \{v_i, v_j, v_k\}}$$

$$\frac{}{\texttt{let}\ x = v\ \texttt{in}\ e \to e[x \mapsto v]}$$

$$\frac{[\![C]\!] = i}{\texttt{if}\ C\ \texttt{then}\ e_1\ \texttt{else}\ e_0 \to e_i}$$

$$\frac{}{\texttt{case:}\ v^1, v^2, v^3 \to \{v_1^1, v_2^2, v_3^3\}}$$

$$\frac{}{\texttt{rngBit}() \xrightarrow{1/8} \{b_1, b_2, b_3\}}$$

Figure 2: Semantics of the core language

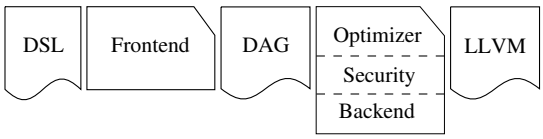


Figure 3: Protocol DSL compiler pipeline

agate bottom values (due to the case-construct introducing them). However, this is not a problem for us because we are compiling to an IR in the form of a finite directed acyclic graph (DAG) with no control flow constructs. This allows us to eliminate all such inefficiencies with dead-code elimination, by throwing away bottom values and the operations that have introduced them. This semantics is the basis for our compiler.

3.3 Compiler implementation

High-level overview of the compilation pipeline is presented in Fig. 3. The frontend performs, in respective order, lexical analysis, syntactic analysis, static checking and translation to the low-level intermediate DAG representation. The IR is optimized, statically checked for security guarantees and compiled to LLVM [29] code. The generated LLVM code can in turn be compiled and linked with Sharemind. The generated code is not tightly coupled to Sharemind and does not depend at all on Sharemind's functionality. Instead the control is inverted, so that the generated code provides meta information that Sharemind reads, and callbacks for every computation round that Sharemind invokes with requested data (received messages, randomly generated data etc.). Many other SMC systems can use the generated code given that they are able to send network messages and provide random numbers.

On the high-level language we perform data type verification and party type verification. After static checks we translate the high-level code to a much simpler code that is based on system F_ω (λ-calculus with type application and type operators). This representation is evaluated to a normal form which is converted to the IR. The IR has well defined syntax and semantics and is optimized with a separate tool. Security analysis is performed on the IR to provide additional guarantees that optimizations preserve security.

3.4 Low-level Intermediate Representation

Arithmetic circuits are the IR for our protocol compiler; this representation is used for optimizations. An arithmetic circuit is a directed acyclic graph (DAG), where the vertices are labeled with operations and the incoming edges of each vertex are ordered. The input nodes of the circuit correspond to the representation of the inputs to the ABB operation that this protocol implements; in case of protocol sets based on secret sharing, each input is represented by a number of nodes equal to the number of the protocol parties. Similarly, the output nodes correspond to shares of the output.

Communication between parties is expressed implicitly: each node of the circuit is annotated with the executing party, and an edge between nodes belonging to different parties denotes communication. Such representation makes both the aspects of computation (relationships between values) and communication (how many bits are sent in how many rounds?) in the protocol easily accessible for analyses and optimizations. The fact that our DAG representation contains no control-flow constructs or loops makes accurate analysis and powerful optimizations possible even on large graphs.

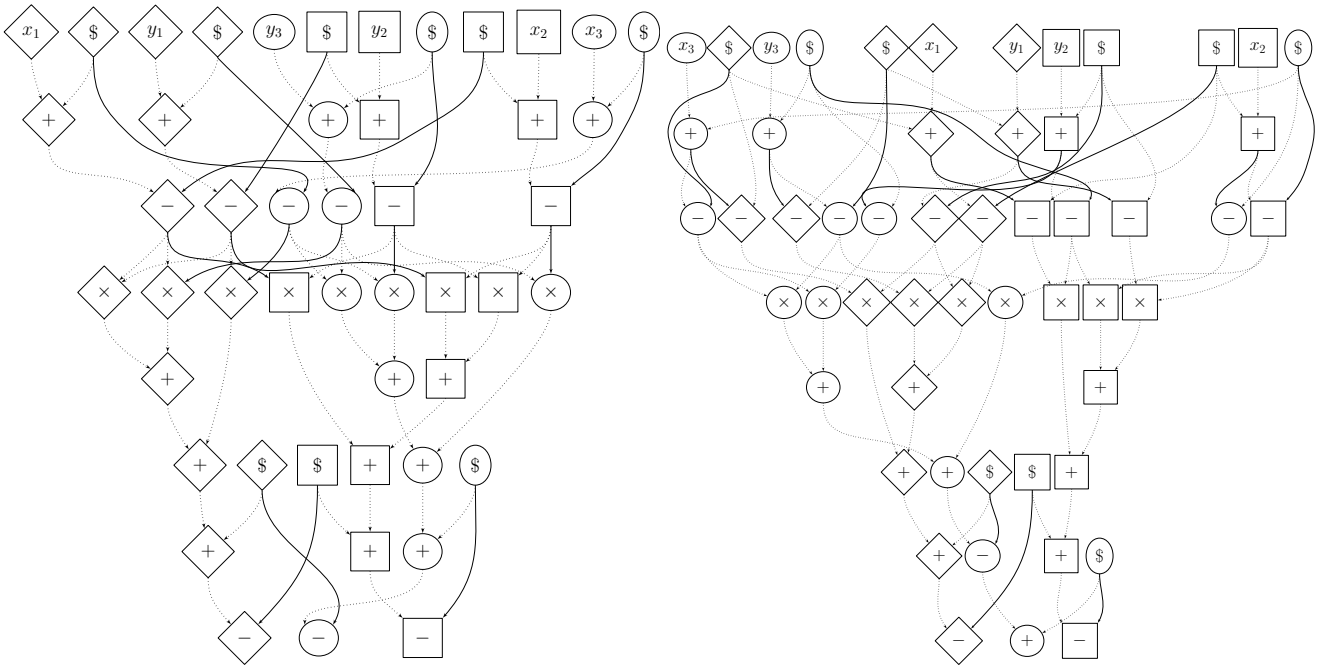
To compile the protocols specified in our protocol DSL to circuits, loops have to be unrolled, function calls inlined, etc. The type system and the compiler of the DSL ensure that loop counts and function call depths (even for recursive functions) are known during compile time. If the control flow of a protocol requires the knowledge of (public) data known only at runtime (e.g. the length of an array), then this protocol cannot be fully specified in the protocol DSL and SecreC [6] has to be at least partially used. The circuit corresponding to the multiplication protocol (Listing 3) is shown in Fig. 4. Different parties are identified by different node shapes. A solid edge denotes communication. We see that this protocol requires two rounds, because there are paths in this graph that contain two solid edges.

3.5 Optimizations

The IR is used to optimize the protocols. Due to the compositional nature of specification, the protocols typically contain constants that can be folded, duplicate computations, dead code, etc. So far, we have implemented all optimizations analogous to the ones reported in [27] for Boolean circuits (constant propagation, merging of identical nodes, dead code removal). But as our circuits are much smaller (the biggest protocols have less than a hundred thousand nodes), and the arithmetic operations allow much more information about the computation to be easily gleaned, we have also successfully run more complex optimizations. We can simplify certain arithmetic expressions, such as linear combinations, even if communication is involved between operations.

Interestingly, we can move certain computations from one party to another, or even duplicate computations, if it results in decrease of communication (which is the bottleneck for current protocols of Sharemind). In the multiplication protocol in Fig. 4(a), we can reduce the number of rounds to 1 by duplicating six subtraction nodes and assigning them to different parties (box → oval → diamond → box) resulting in the circuit in Fig. 4(b). This does not turn a secure protocol insecure because it does not make the view of any party richer than it was. A small downside of the optimization is that the resulting circuit has more nodes and thus has become slightly slower to evaluate. This is an exception and in most cases the optimizer reduces the size of circuits by a significant margin.

To analyze the reduction in communication we have to view the total communication as a sum of two parts: online and offline. The online part consists of communication from nodes that depend on the inputs of the protocol. The offline part depends only on the randomly generated values and is a significant portion of overall communication: the extensively used resharing protocols and share conversion protocol generate such nodes. The circuit optimizer manages to reduce the amount of online communication in most of the protocols by a sizable amount (up to 10%) but does not change the total communication. Table 1 lists the ratios for node counts and online communication for unoptimized and optimized



(a) Generated DAG (b) Optimized DAG

Figure 4: Multiplication protocol circuit

protocols. The table contains ratios for the following operations: integer multiplication (`mul`), bit-shift right by a private (`shr`) and by a public (`shrc`) amount, division with private (`div`) and public divisor (`divc`), floating-point addition `fadd` and multiplication `fmul`, floating-point reciprocal `inv`, square root `sqrt` and natural logarithm `ln`. Integer operations have only been measured for 64-bit integers and floating-point operations only for single-precision operations. The size ratio of 1.12 for multiplication protocol means that the optimizer added nodes.

uint64	`mul`	`shrc`	`shr`	`divc`	`div`
Size ratio	1.12	0.84	0.56	0.90	0.81
Online ratio	1.00	0.96	0.99	0.97	0.97
float32	`fadd`	`fmul`	`inv`	`sqrt`	`ln`
Size ratio	0.77	0.86	0.84	0.86	0.79
Online ratio	0.90	0.95	0.98	0.93	0.91

Table 1: Optimizer performance: ratio of code size and online communication

3.6 Integration with Sharemind

The specified protocols are used to generate protocol implementations for the Sharemind platform. While the specifications are usually polymorphic in bit-width of the arguments and the result, the Sharemind protocols work with integers of fixed length. Hence, together with our protocols we also specify the input and output widths for which we want the implementations to be generated. E.g., Sharemind currently has protocols for multiplying 8-, 16-, 32-, and 64-bit integers.

The protocols are first translated to a low-level IR, which is then compiled to LLVM code. The protocols in Sec. 2 are defined for scalar values, but Sharemind mostly operates on vectors of values and thus expects that operations are vectorized. Hence the LLVM code generation also performs automatic vectorization (this process does not increase the round count) — the inputs to the protocol are vectors of values, which are pointwise operated on, and result in a vector of outputs. The vectorization step is very important because network latency is orders of magnitude larger than the time to takes to perform arithmetic operations. E.g. it is an order of magnitude faster to run the multiplication protocol on thousand elements in parallel than it is do the same thing iteratively.

In code generation, the necessary communication between parties is derived from the accesses to a party's values from a different party's code, which is directly visible in the DAG. Communication is realized with the help of Sharemind's networking API, packing all values communicated at the same round into a single message or a few messages of suitable length. During the translation to the IR, all polymorphism is resolved, hence each compiled protocol is used with values of a particular length and lengths of all exchanged messages are known at compile-time. The DSL generated protocols are called through high-level language scripts. High-level SecreC [6] code is translated to bytecode that the Sharemind virtual machine evaluates. The bytecode specifies the control flow, the invoked protocols, and their arguments. The actual invocations are performed by the virtual machine. This is summarized in Fig. 5.

4. SECURITY

4.1 Security definitions

The three-party additive secret-sharing based protocol set that Sharemind uses provides security against one honest-but-curious party. Security is defined as the indistinguishability of the actual execution of the protocol from a simulated one. Security implies that (i) the protocol preserves the *privacy* of honest parties' inputs, and (ii) the protocol delivers correct outputs to all honest parties. For honest-but-curious adversaries, the second property trivially

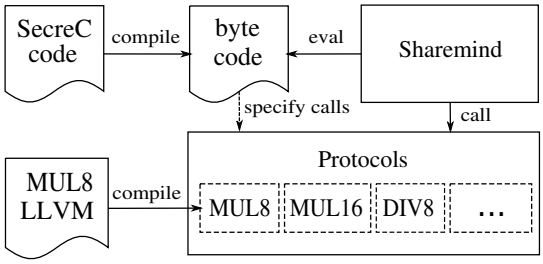


Figure 5: Sharemind architecture

holds. We have attempted to construct the protocols of SHAREMIND in such a way that they would *provide privacy* [5, 37] against one *malicious* computing party.

We use the *universal composability (UC)* [11] framework to argue about the security and privacy properties of our protocol set. In this framework, a protocol — a *real functionality* — is modeled as a collection of interactive Turing machines (ITM). These machines communicate with each other over a number of named tapes. In a *closed collection*, each named tape has a single machine writing it, and a single machine reading it. Only closed collections of ITMs are executable. The *interface* of a collection of ITMs is the set of named tapes occurring in it that lack a reader or writer. In the UC framework, the interface of a functionality is split into two parts — the interface for the intended user (called *the environment*) of the functionality, and the interface for the adversary.

Let π be a real functionality (a protocol — a collection of ITMs) and $\mathcal{F}$ an ideal functionality (usually expressed as a single ITM). We say that π is *black-box at least as secure as* $\mathcal{F}$, if there exists an ITM Sim, such that all environments $\mathcal{Z}$ and all adversaries $\mathcal{A}$, the views of $\mathcal{Z}$ in closed collections $\mathcal{Z}\|\pi\|\mathcal{A}$ and $\mathcal{Z}\|\mathcal{F}\|(\mathsf{Sim}\|\mathcal{A})$ are indistinguishable [11]. Here $\|$ denotes the parallel composition of ITM-s or their collections, together with the identification of tapes with the same name.

If π is an SMC protocol for performing a particular operation op on shared values, then the ITMs of π receive the shares of the inputs over the interface with $\mathcal{Z}$ at the beginning of the protocol. Also, the adversary may *corrupt* some ITMs at the beginning of the protocol. During the protocol, the ITMs exchange messages and compute the shares of the outputs. Corrupt ITMs send everything they receive also to the adversary. If the adversary is malicious, then the corrupt ITMs do not follow the protocol but the adversary's orders. Finally, the ITMs of π hand the shares of the outputs back to $\mathcal{Z}$. A corresponding ideal functionality $\mathcal{F}_{\mathsf{op}}$ also receives the shares of the inputs from $\mathcal{Z}$ and the corruption requests from $\mathcal{A}$. The functionality $\mathcal{F}_{\mathsf{op}}$ reconstructs the actual inputs from the shares, applies op to them and shares the results, thereby obtaining the output shares which it gives back to $\mathcal{Z}$. The functionality $\mathcal{F}_{\mathsf{op}}$ also sends the input and output shares of corrupt parties to $\mathcal{A}$. If the adversary is malicious, then it can also change the output shares returned to corrupt parties.

An SMC protocol π for the operation op is *secure* if it is at least as secure as the corresponding ideal functionality $\mathcal{F}_{\mathsf{op}}$. E.g. the resharing protocol in Listing 1 is a secure protocol for the identity function. The privacy of a protocol can be defined similarly. Let the protocol $\bar{\pi}$ be obtained from π and the functionality $\bar{\mathcal{F}}$ be obtained from $\mathcal{F}_{\mathsf{op}}$ by removing from them the final sendings of output shares to $\mathcal{Z}$ and $\mathcal{A}$. We say that π is *private* if $\bar{\pi}$ is at least as secure as $\bar{\mathcal{F}}$.

It is known that the sequential composition of two protocols (where the output shares from the first protocol are directly given as inputs to the second protocol, without going through $\mathcal{Z}$) preserves privacy [5]. Against honest-but-curious adversaries, the composition of a private protocol and a secure protocol (commonly resharing) is secure [5]. Against malicious adversaries, privacy limits the amount of information they can learn about the inputs of the protocol by interfering with it [37].

A protocol optimization $\mathcal{T}$ transforms a protocol π implementing an ideal functionality $\mathcal{F}$ into a new protocol $\mathcal{T}(\pi)$ also implementing $\mathcal{F}$. Each optimization we have implemented in our DSL compiler *preserves the preservation of privacy* — we show that there exists an ITM T, such that

$$\forall \mathsf{Sim} : [\forall \mathcal{Z}, \mathcal{A} : \mathcal{Z}\|\pi\|\mathcal{A} \approx \mathcal{Z}\|\mathcal{F}\|(\mathsf{Sim}\|\mathcal{A}) \Rightarrow \forall \mathcal{Z}, \mathcal{A} : \mathcal{Z}\|\mathcal{T}(\pi)\|\mathcal{A} \approx \mathcal{Z}\|\mathcal{F}\|((\mathsf{T}\|\mathsf{Sim}_\alpha)\|\mathcal{A})],$$

where Sim_α denotes Sim with renamed tapes, such that it directly communicates only with T. In effect, if $\mathcal{T}$ is a protocol transformation, then $\mathsf{T}\|(\cdot)_\alpha$ is the corresponding *simulator transformation* used to transform a security proof of π to a security proof of $\mathcal{T}(\pi)$.

4.2 Proving security of protocols

Our protocol DSL contains no particular mechanisms to statically ensure the security or privacy of protocols. The type system of the language only controls the lengths of the values, but not their dependence on inputs or random variables. Hence we also cannot speak about security-preserving compilation in the sense of [19].

We ensure the security of the generated protocols using data flow analysis at the level of the IR. Our compiler pipeline contains the static analyzer by Pettai and Laud [37] that checks protocols expressed as arithmetic circuits for privacy against malicious adversaries. If a protocol passes that check and we know that it is followed by a resharing protocol, then it is also secure against honest-but-curious adversaries. The check is invoked after the translation from the protocol DSL to the intermediate language, and the optimization of the generated intermediate code. Having the security check late in the pipeline ensures that the earlier operations do not introduce uncaught vulnerabilities.

Our experience with the protocol DSL validates the security aspects of the language and compiler design. Indeed, we have found the writing of secure protocols to be very straightforward. This can be explained by most of the protocols being written as a composition of simpler ones. When writing in this style the composition theorem automatically provides the privacy guarantee. It is very rare that a protocol is added that is not purely a composition, and even in this case the automatic privacy checker is there to provide a safety net and validation for the programmer. In fact, for those reasons, to implement efficient and secure protocols in the protocol DSL one does not need to have a deep understanding of the security framework of the additive secret sharing scheme. The protocol implementor never has to show that an implemented protocol is secure: it is often trivially so by virtue of composition and this fact is always automatically verified regardless.

5. EXPERIMENTAL RESULTS

We have implemented protocols for integer operations from [8] and for floating-point operations from [23]. In this section we explore the performance of some of these DSL implemented protocols compared to the existing C++ protocol set. In most cases the evaluated protocols are not algorithmically identical. This is unavoidable, as a key design point of the DSL is to simplify the optimization and exploration of protocols. The protocols in C++ are long (the C++ division protocol spans over 1500 lines of code whereas all DSL protocols combined span less than 3000) and difficult to read. It is often not clear if their implementation matches

the specification – and frequently it does not as the concrete implementations tend to employ undocumented optimizations.

Another factor that makes identical comparison difficult is that the DSL floating-point protocols provide better (accuracy) guarantees. For instance, we found out that some of the C++ protocols do not handle 0 properly, some operations have poor relative errors, and double-precision floating-point numbers provide very poor accuracy guarantees (only in the range of 10^{-7}). Providing fair comparison would either mean incorporating those defects into DSL protocols or improving the C++ protocols. In both cases valuable time is wasted. The poor accuracy guarantees of C++ protocols are due to double-precision operations requiring the use of larger than 128-bit integers which the old framework had difficulties with. These differences give a performance advantage to C++ protocols as they do not handle some of the cases and do not operate on integers as large as those used in the new protocols.

Experiments were performed on a cluster of 3 identical computers connected with a 10 Gigabit Ethernet network. Each computer was equipped with 128GB DDR4 RAM, two 8 core Intel Xeon (E5-2640 v3) processors and was running Debian Jessie (14th May 2015). Every protocol was benchmarked on various numbers of inputs: when executing a protocol in parallel on multiple inputs, the round count remains the same while the amount of network communication increases. On a decent network connection, the evaluation of the multiplication protocol on scalars, and on vectors of length 10000, takes roughly the same time.

On each input length we have evaluated every protocol at least twenty times; and up to few hundred times on smaller input sizes in order to reduce variance. To estimate the execution time of a protocol on some input length, we computed the mean of all measurements on that length. The *speedup* was computed by dividing the estimated execution time of the old protocol with the estimated execution time of the respective DSL generated protocol. Speedup greater than one means that the new protocol was, on average, faster than the old one. All of the measurements were performance in an identical setup, using the same unmodified software versions.

In Fig. 6(a) we can see the running time of the DSL and C++ floating-point multiplication protocols depending on the input size. Notice how the running time is roughly constant up to around 500 elements after which the execution time grows linearly. We call this point a *saturation point* because from this point on the execution time is no longer latency bound and is limited by some other factors (such as bandwidth or computation). Fig. 6(b) shows the speedup of the protocol compared to the C++ version. We can see that the new floating-point multiplication is between two to six times faster.

Complete benchmarking results are presented in Table 2. The integer operations have been benchmarked on 32-bit integers and floating-point protocols on single-precision numbers. We chose to display only 32-bit versions because 64-bit integer division protocols are not implemented in C++ and for the rest of the operations the results are quite similar, mostly favouring DSL protocols. The shift-right with a private value ($a \gg b$) protocol has benefited a lot from a redesign in the DSL.

We can see that in every case the protocol DSL provides us an improved floating-point arithmetic operation as we can expect the protocols to run at least twice as fast. In many cases the DSL provides up to 10 times faster protocols. Some slowdown with integer division operations was to be expected as they are very well tuned medium-sized protocols. Still, past 10-element inputs we can see some speedup in favour of the DSL. Integer multiplication is provided here as a pathological worst case for us: multiplication is a very small and simple protocol that is easy to hand-tune and thus we should not expect the DSL generated version to compete. This is exactly so and we see a drastic drop in performance past 10^4-element input vectors. We do not have a full explanation for this drop, but one possibility is that the current multiplication protocol continues to compute intermediate values while some network messages are being sent. The hand-tuned implementation might also send messages in a more suitable pattern for the Sharemind's network layer.

	10^0	10^1	10^2	10^3	10^4	10^5
$a \times b$	1.12	0.96	0.90	1.05	1.10	0.68
$a \gg n$	0.90	1.07	1.36	2.36	2.11	1.32
$a \gg b$	2.13	4.31	13.56	28.35	35.97	27.99
a / n	0.78	0.73	1.54	2.65	2.67	3.03
a / b	0.90	1.02	1.33	1.27	1.05	0.95
$x + y$	2.62	2.87	3.65	4.38	3.93	3.79
$x \times y$	1.74	1.89	3.28	4.21	4.18	4.37
1 / x	1.91	2.42	3.79	3.95	3.95	3.90
$\sqrt{x}$	2.50	3.33	5.01	5.81	5.81	5.96
$\sin x$	9.23	10.62	13.10	12.62	9.50	
$\ln x$	12.29	12.68	14.32	11.12	7.83	
e^x	5.33	6.55	10.47	11.48	10.97	
erf x	4.71	12.32	29.02	36.32	41.73	

Note: a and b denote 32-bit additively shared integers; n denotes 32-bit public integer; x and y denote single-precision private floating-point numbers.

Table 2: Speedup in comparison to non-DSL protocols

We have also benchmarked private satellite collision analysis from [23] using the new protocols. We see roughly 5-fold speedup, going from 0.5 satellite pairs per second to 2.5 pairs per second. When processing 100 pairs in parallel we gain a roughly 8-fold speedup going from processing 0.7 pairs per second to 6 pairs per second. This demonstrates that improving low-level floating-point protocols can have a great effect on high-level applications.

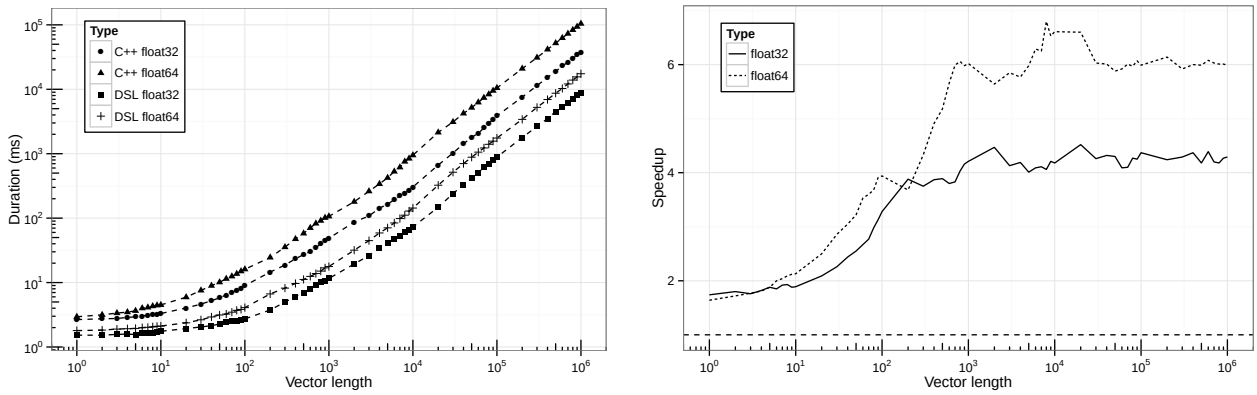
6. RELATED WORK

A fair number of languages for SMC have been proposed over the years, aiming to simplify the implementation of SMC protocols, and to allow the developer to concentrate on application logic. Most of the languages concentrate purely on application logic, expecting the SMC runtime to invoke a specific protocol for each operation on private values that occurs in the program. Such languages include SecreC [6] used in the Sharemind framework, and SFDL, compiled into Boolean circuits in the FairplayMP framework [2] based on garbled circuits [42].

In several systems, the privacy-preserving application is expressed in some widely used programming language, possibly with some restrictions and privacy-related annotations. The program is analyzed and operations on private data replaced with calls to the implementations of protocols for these operations. The resulting program can be compiled, and results in an executable, distributed application making use of SMC. The language used may be C [43], Java [40], Python [24] or Haskell [35]. Alternatively, the program may be translated into Boolean circuits [21], which are then optimized [26] and garbled.

A number of proposed languages can express both the computations performed through SMC protocols, and the computations performed privately by each participant [36, 20, 39, 33, 38]. They are not meant, nor capable to describe the details of those protocols.

None of the languages described above are helpful in stating how the low-level SMC protocols are implemented. As far as we



(a) Running time

(b) Speedup of DSL compared to C++

Figure 6: Benchmarking results for floating-point multiplication

know, only Launchbury et al. [30] seriously discuss the structuring and higher-level constructions for these protocol implementations. They do not propose a novel language, but use Haskell with a primitive operation `rotateRight` (corresponding to our **`from Next`**). Their approach does not straightforwardly allow the parallelization of protocols to the same extent as ours.

Our DSL has some similarities to dataflow programming [18], where the communication pattern follows from the data dependencies in the program. Our **`from`**-primitive is slightly similar to the concept of a *future* [1]. Still, the relationship with these paradigms is not too close, because for us, the concept of a party is also fundamental, and we explicitly state which computations are executed by which parties. Hence we believe that the design of a DSL for a domain similar to ours (building complex protocols between mutually distrustful parties) has not been considered before.

7. CONCLUSIONS

We have presented a DSL for specifying low-level SMC protocols. The ultimate goal of this DSL is to improve the efficiency of different kinds of SMC applications, thereby facilitating the adoption of this technique. The DSL achieves this goal by increasing the efficiency of executable protocols, easing their development and maintenance, and simplifying the comparison of different design decisions for protocols.

At least the following aspects of its design contribute to the success of our DSL: (i) it is separate from the application-level language; (ii) it concentrates on describing the data dependencies of a protocol, not the computation and communication details; (iii) its communication primitives are tailored to the needs of low-level SMC protocols; (iv) its type system is length-polymorphic, yet allows precise control over the lengths of input, output, and intermediate values; (v) its intermediate representation (IR) is extremely parallelization-friendly; (vi) the IR consists of arithmetic operations; (vii) the optimizations targeting the IR are applied to the whole protocol; (viii) the optimizations preserve the security of protocols; and (ix) the security checker is invoked late in the toolchain. These fortunate aspects build upon and strengthen each other. E.g. if there was a single language for primitive protocols and applications, resulting in a monolithic protocol for the entire privacy-preserving application, then the whole-protocol optimizations would be infeasible. The use of arithmetic circuits, as opposed to Boolean ones, also keeps down the size of the protocol description. The compositional nature of protocols, together with the security-preserving optimizations, ease the development of secure protocols, and the security checker instills confidence.

We believe that it is worthwhile to use the same or similar language for developing the low-level protocols in SMC frameworks based on Shamir's secret sharing [43] or SPDZ [24]. Similar improvements in efficiency and maintainability would be obtained.

Our DSL brings structure to the implementations of low-level protocols and opens up new optimization possibilities for the application level language. While the application language already supports SIMD operations, we can now add statement-level parallelism to it. We can also estimate the performance of low-level protocols much more precisely, depending on the parameters of the execution environment of the SMC application, so that the application compiler can better choose the applied optimizations.

8. ACKNOWLEDGEMENTS

This work was supported by the European Social Fund through the ICT Doctoral School programme, and by the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS, and through the Software Technologies and Applications Competence Centre, STACC. It has also received support from Estonian Research Council through project IUT27-1. We would like to thank Madis Janson, Liisi Kerik, Alisa Pankova, Martin Pettai and Karl Tarbe for their contributions to the compiler and protocols.

9. REFERENCES

[1] Henry C. Baker, Jr. and Carl Hewitt. The Incremental Garbage Collection of Processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59. ACM, 1977.

[2] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266, New York, NY, USA, 2008. ACM.

[3] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. A Short Paper on How the National Tax Office Evaluated a Tax Fraud Detection System Based on Secure Multi-party Computation. In *Proceedings of 19th International Conference on Financial Cryptography and Data Security*, 2015.

[4] Dan Bogdanov, Liina Kamm, Sven Laur, and Ville Sokk. Rmind: a tool for cryptographically secure statistical analysis. Cryptology ePrint Archive, Report 2014/512, 2014.

[5] Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. From input private to universally composable secure multi-party computation primitives. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014*, pages 184–198. IEEE, July 2014.

[6] Dan Bogdanov, Peeter Laud, and Jaak Randmets. Domain-polymorphic programming of privacy-preserving applications. In Alejandro Russo and Omer

Tripp, editors, *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, Uppsala, Sweden, July 29, 2014*, page 53. ACM, 2014.

[7] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[8] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012.

[9] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[10] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, pages 223–239, Washington, DC, USA, 2010.

[11] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[12] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, January 25-28, 2010, Revised Selected Papers*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2010.

[13] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[14] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.

[15] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

[16] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[17] Fabienne Eigner, Matteo Maffei, Ivan Pryvalov, Francesca Pampaloni, and Aniket Kate. Differentially private data aggregation with optimal utility. In Charles N. Payne Jr., Adam Hahn, Kevin R. B. Butler, and Micah Sherr, editors, *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, pages 316–325. ACM, 2014.

[18] Jim Falgout. Dataflow Programming: Handling Huge Data Loads Without Adding Complexity. *Dr. Dobb's Journal*, 36, 9 2011.

[19] Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 432–441. ACM, 2009.

[20] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security. CCS'10*, pages 451–462. ACM, 2010.

[21] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 772–783. ACM, 2012.

[22] Liina Kamm, Dan Bogdanov, Sven Laur, and Jaak Vilo. A new way to protect privacy in large-scale genome-wide association studies. *Bioinformatics*, 29(7):886–893, 2013.

[23] Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, pages 1–18, 2014.

[24] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 549–560. ACM, 2013.

[25] Florian Kerschbaum, Axel Schröpfer, Antonio Zilli, Richard Pibernik, Octavian Catrina, Sebastiaan de Hoogh, Berry Schoenmakers, Stelvio Cimato, and Ernesto Damiani. Secure collaborative supply-chain management. *IEEE Computer*, 44(9):38–43, 2011.

[26] Benjamin Kreuter, Abhi Shelat, Benjamin Mood, and Kevin R. B. Butler. PCF: A portable circuit format for scalable two-party secure computation. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 321–336. USENIX Association, 2013.

[27] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 285–300. USENIX Association, 2012.

[28] Toomas Krips and Jan Willemson. Hybrid model of fixed and floating point numbers in secure multiparty computations. In Sherman S. M. Chow, Jan Camenisch, Lucas Chi Kwong Hui, and Siu-Ming Yiu, editors, *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*, volume 8783 of *Lecture Notes in Computer Science*, pages 179–197. Springer, 2014.

[29] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[30] John Launchbury, Iavor S. Diatchki, Thomas DuBuisson, and Andy Adams-Moran. Efficient lookup-table protocol in secure multiparty computation. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 189–200. ACM, 2012.

[31] Jeff Lewis. Cryptol: specification, implementation and verification of high-grade cryptographic applications. In Peng Ning, Vijay Atluri, Virgil D. Gligor, and Heiko Mantel, editors, *FMSE*, page 41. ACM, 2007.

[32] Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011.

[33] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating efficient ram-model secure computation. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 623–638. IEEE Computer Society, 2014.

[34] Lior Malka. Vmcrypt: modular software architecture for scalable secure computation. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 715–724. ACM, 2011.

[35] John C. Mitchell, Rahul Sharma, Deian Stefan, and Joe Zimmerman. Information-flow control for programming on encrypted data. In Stephen Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 45–60. IEEE, 2012.

[36] Janus Dam Nielsen and Michael I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In Michael W. Hicks, editor, *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security. PLAS'07*, pages 21–30. ACM, 2007.

[37] Martin Pettai and Peeter Laud. Automatic Proofs of Privacy of Secure Multi-Party Computation Protocols Against Active Adversaries. In Cedric Fournet and Michael Hicks, editors, *2015 IEEE 28th Computer Security Foundations Symposium (CSF 2015)*, 2015.

[38] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 655–670. IEEE Computer Society, 2014.

[39] Axel Schröpfer, Florian Kerschbaum, and Guenter Mueller. L1 - An Intermediate Language for Mixed-Protocol Secure Computation. In *Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference. COMPSAC'11*, pages 298–307. IEEE Computer Society, 2011.

[40] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd D. Millstein. Mrcrypt: static analysis for secure cloud computations. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 271–286. ACM, 2013.

[41] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X) Modular type inference with local assumptions. *Journal of Functional Programming*, 21(4-5):333–412, 2011.

[42] Andrew C. Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press.

[43] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: a general-purpose compiler for private distributed computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 813–826. ACM, 2013.