Efficient Lookup-Table Protocol in Secure Multiparty Computation

John Launchbury, Iavor S. Diatchki, Thomas DuBuisson, Andy Adams-Moran

Galois, Inc. *

{john, diatchki, tommd, adams-moran}@galois.com

Abstract

Secure multiparty computation (SMC) permits a collection of parties to compute a collaborative result, without any of the parties gaining any knowledge about the inputs provided by other parties. Specifications for SMC are commonly presented as boolean circuits, where optimizations come mostly from reducing the number of multiply-operations (including *and*-gates)—these are the operations which incur significant cost, either in computation overhead or in communication between the parties. Instead, we take a language-oriented approach, and consequently are able to explore many other kinds of optimizations. We present an efficient and general purpose SMC table-lookup algorithm that can serve as a direct alternative to circuits. Looking up a private (i.e. shared, or encrypted) n-bit argument in a public table requires $log(n)$ parallel-and operations. We use the advanced encryption standard algorithm (AES) as a driving motivation, and by introducing different kinds of parallelization techniques, produce the fastest current SMC implementation of AES, improving the best previously reported results by well over an order of magnitude.

Categories and Subject Descriptors D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications: Specialized application languages

Keywords Secure Multiparty Computation, Cloud, EDSL, Haskell

1. Introduction

There is growing interest in performing computation on encrypted data, partly motivated by the challenges of cloud computing. As we lose control of the *location* of our data, we still want to retain control of the *confidentiality* and/or *integrity* of our data. If we could encrypt our data (either for confidentiality, or for integrity) and then have the cloud operators perform computations on the data in the encrypted form, then we may have the best of both worlds: the cloud supplies storage and computational resources, while the encryption provides guarantees about what happens to the data.

At first it seems quite improbable that it would be possible to perform computations on encrypted data. After all, once the data is encrypted, it is completely obscured. However, the cryptography community has long known that some kinds of computations are possible—at least in principle. This was notably demonstrated by Yao's seminal work on secure multiparty computation (SMC) [Y86], and more recently by Gentry's work on fully homomorphic encryption (FHE) [G09]. SMC computations permit a collection of parties to compute a collaborative result, without any of the parties gaining any knowledge about the inputs provided by other parties (at least, nothing more than would be derivable from the final result of the computation).

SMC protocols can be targeted to different security models, but the performance cost in establishing and maintaining the security for particular models can vary significantly. The simplest security model used for SMC is *honest but curious* [G04], where the separate parties are assumed to follow the protocol honestly, but may at the same time attempt to learn secrets by looking at internal values of the computation, including any communications. This security model is appropriate for settings such as preventing information leakage by individuals with administrator access, or after a cyber break-in. There are also fairly generic techniques for augmenting honest-but-curious protocols to provide more stringent security guarantees (such as against malicious adversaries who intend to subvert the computation), so the honest-but-curious protocol may be seen as a significant first step towards constructing more secure versions.

Honest-but-curious was exactly the security model considered sufficient for a Danish beet auction in 2008 [BCD+08]. There, 1200 Danish farmers submitted obfuscated bids to three servers that were run by distinct agencies. Each of the agencies was considered well motivated to follow the multi-party protocols honestly, and the confidentiality built into the SMC protocols provided sufficient reassurance to the farmers, 78% of whom agreed that, "it is important that my bids are kept confidential."

While Gentry-style FHE is very new and still wildly impractical, there has been significant effort in the last 10 years or so to make SMC usable in practice. The fact that a genuine commercial auction could be programmed in this style is a testament to the progress made. However, the state of the art is such that an SMC implementation of an algorithm such as the AES block cipher [NIST01] is still considered challenging, with execution times of around a few blocks per second (very, very slow compared with non-secure computation). There are two fundamental reasons for this: first, all SMC computations have to be performed generically across all possible input and internal values (otherwise information is revealed), and second, SMC schemes require significant network

* This material is based upon work supported by the Defense Advanced Research Projects Agency through the U.S. Office of Naval Research under Contract N00014-11-C-0333. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

ICFP'12, September 9–15, 2012, Copenhagen, Denmark.

communication, typically growing linearly with the number of and-gates in the function being evaluated.

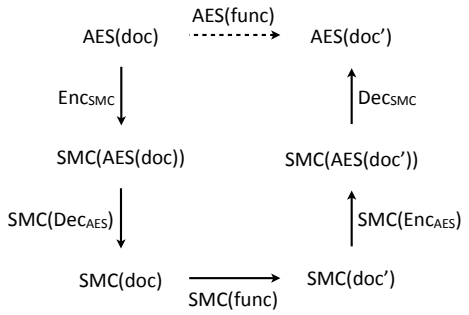

Figure 1. Lifting Secure Computation to AES

The AES block cipher is also interesting in its own right as a part of the SMC universe. Many use cases for computation on encrypted data will end up using AES to optimize one or more aspects of the data flow. Here are two example scenarios:

- A user records video on a smartphone and wants to upload it to the cloud for storage and processing (lighting adjustment, facial recognition, etc.). Encrypting the video stream directly using an encrypted computation scheme such as SMC or FHE would incur a significant communication bandwidth overhead (e.g. anything from a factor of three to factors of thousands, depending on the scheme). It would be much more desirable to encrypt the video with AES to upload it to the cloud, and then securely decrypt the video-stream as part of the SMC processing.
- Data previously stored in the cloud is more likely to be encrypted with a standard block cipher such as AES than with any given SMC scheme. Therefore, many SMC applications are likely to start with AES decrypt, perform the computation in the SMC context, and conclude with an AES encrypt to place the processed result back in the cloud.

Figure 1 shows how to lift the capability for secure computation to AES-encrypted documents. Assuming that we know how to compute the bottom arrow (i.e. that we know how to securely evaluate some function *func* with SMC), if we can also securely evaluate AES encryption and decryption, then we have created a scheme for securely computing *func* in an AES-encrypted context. Note that in this setting it makes no sense to perform the AES decryption and/or encryption in the clear, otherwise the data it is protecting would be revealed.

1.1 Contributions of this paper

The prevailing approach to SMC is to express the computation to be performed as a circuit, and then to process that circuit with cryptographic techniques. Efficiencies are gained first by optimizing the translation and processing, and then by staging the phases of circuit generation so that only portions of the circuit exist at any time.

An alternative view of SMC is to view the secure multiparty computational substrate as an *SMC-machine*—an abstract machine with highly non-standard interface and performance properties. Efficiencies are gained through minimizing expensive operations, by reducing the overheads of individual operations (through exploiting opportunities for SIMD-like parallelization if available), and by hiding residual latencies involved in network-based operations. This language-centric approach is what we describe in this paper. Our particular contributions are as follows:

- We present an innovative SMC protocol for performing table lookups. In a RAM-machine table-lookup is trivial. Not so on a SMC-machine. Our table lookup protocol minimizes the number of parallel-ands required, and provides a general mechanism for compiling many kinds of functions into a (SIMD-parallelizable) SMC framework in a regular way—any n-bit input function can be represented as a lookup table of size 2^n. The protocol presented here performs an n-bit table lookup using $log(n)$ parallel-and operations (i.e. bit-wise multiply), where each *and*-operation operates on up to 2^n individual bits.

  If n remains small enough (say below about 20), then SMC schemes may perform better by using these table lookups rather than attempting to evaluate a function directly. That is, it may be more efficient to have a table storing the pre-computed results of the function and just looking up the answer. SMC computations are often limited by the cost of communication, but with table lookup the limiting factor is most likely to be the local size of the lookup tables rather than the size or number of communications. The reason is that the size of communication is one bit per table entry, with the number of communications being the log of the log of the number of table entries.
- Using the table-lookup protocol, we present an SMC implementation of AES that is dramatically faster than anything previously reported. Depending on the specific machine configurations, we achieve speeds of over 300 AES blocks per second, compared with other SMC schemes, the fastest of which previously reported is around 17 AES blocks per second[1].

  Unlike other approaches to doing AES in SMC, we do not rely on the algebraic properties of the AES algorithm itself, but adapt a standard table-driven algorithm that is popular in both software and hardware. The table-driven approach treats the internal lookup tables of AES (called *S-boxes*) as unstructured, so the techniques we develop here carry over directly to other applications.
- To enable each of these contributions, we developed a library for SMC in Haskell which, in effect, provides an embedded domain-specific language (EDSL) for programming SMC applications. The EDSL allows us to explore multiple execution tradeoffs. In particular, we exhibit the importance of three different kinds of optimizations:
  - Structural algorithmic parallelization within the definition of SMC protocols such as table lookup;
  - Local SIMD parallelism when using specific SMC protocols; and
  - Task-level parallelism to achieve a pipelining effect.

  Interestingly, SMC implementations that use simple secret-sharing schemes are typically presented as either arithmetic sharing, or XOR sharing. Because we construct the EDSL in a type-directed way, we demonstrate that these two styles arise as endpoints in a spectrum of sharing mechanisms that naturally combine both styles.

These contributions are presented specifically in the context of a simple sharing SMC scheme, though the techniques appear to be transferrable to other SMC and FHE schemes. We provide a brief discussion of this in the conclusion.

[1] Of course, to put this in perspective we note that openSSH on similar machines computes over 7 million AES blocks per second. But the comparison would not be fair as each of our AES encryptions requires tens of thousands of openSSH AES encryptions to secure the networking, on top of tens of thousands of native AES encryptions for random number generation. It does, however, give an indication of how far SMC still has to go.

2. Background

For concreteness, we present the lookup-table protocol in the context of a particular sharing scheme, but the protocol is relatively independent of the scheme itself. We assume some familiarity with Haskell, which we use as a notation for protocols.

The SMC scheme we use is simple arithmetic/xor sharing across three peer machines acting as the compute servers. For the protocols we discuss, the three machines run the same code as each other, and communicate (and hence synchronize) between themselves in a cyclic pattern, as shown in Figure 2. Some more complex protocols require less uniform computation and communication patterns, but we won't need them here.

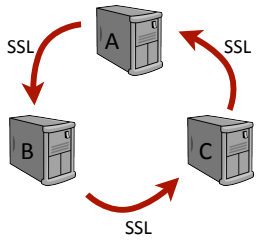

Figure 2. Machine Configuration

In a simple sharing scheme, private (secret) values never exist concretely but instead are represented by three separate *shared* values, each of which lives on one of the peer servers. A value is shared between the machines in a form that is dependent on its type. Fixed-width integer types (e.g `Int16`, `Int32`, etc) are shared *arithmetically*. Thus, a true value x in `Int16` will be shared as three randomly drawn values x_A, x_B, x_C such that $x = x_A + x_B + x_C (mod\ 2^{16})$. The initial shares can be produced by generating x_A and x_B randomly from a uniform distribution, and then defining $x_C = x - x_A - x_B$. Despite x_C being computed, all three numbers exhibit the properties of being fully random, and knowledge of any two of the numbers provides absolutely zero knowledge about the original private value—not even a single bit. Subsequently, the computational protocols will maintain this semantic share property through the calculations that are performed.

Sharing is lifted to structured types as follows: tuples of private values are shared component-wise, and fixed-length sequences of values (i.e. lists or arrays) are shared element-wise. Thus, a private value $[x, y, z]$ will be shared as three randomly drawn values $[x_A, y_A, z_A], [x_B, y_B, z_B], [x_C, y_C, z_C]$ such that $x = x_A + x_B + x_C$, and so on. Sequences of bits are a special case of more general sequences. They need to be handled in an efficient way (else the overhead can kill many algorithmic improvements), so we treat fixed-width bit-vectors (represented as unsigned integers in our library) as if they were sequences of individual bits (i.e. elements of `Int1`, where multiplication is just *and*, and addition is *xor*). Thus, a private value x in `Word8` (a bit-vector of length 8) will be shared as three randomly drawn values x_A, x_B, x_C such that $x = x_A \oplus x_B \oplus x_C$ (where $\oplus$ is bitwise xor).

2.1 Share Operations

To represent the code (i.e. protocol) that runs on each of the machines, we introduce the `Protocol` type (Figure 3). `Protocol` is a variant of the `IO` type, and comes with built-in information about how to communicate with the neighbors. Like `IO`, `Protocol` is a monad, so we can write sequences of operations using the `do`-notation.

Composite protocols are built from primitive protocol operations, which have distinct behaviors for each different value type.

```
type Protocol
instance Monad Protocol

class Entropy a where
  entropy :: a -> Protocol a
  (.+.) :: a -> a -> a
  (.-.) :: a -> a -> a
  (.*.) :: a -> a -> a
```

Figure 3. Primitive Share Operations

We use the type class `Entropy` to overload the protocol operations. The class name `Entropy` reflects the need to access randomness in order to share values. The `entropy` operation returns a random value (fresh entropy) of the same size and shape as its argument[2]. The `entropy` operation is a state-changing operation, so its result type is in the `Protocol` monad. The `.+.` operation is the structural arithmetic operation described above, `.-.` is its inverse, and `.*.` is the corresponding (local) multiplication i.e. multiplication modulo the word size on integer types, lifted pointwise over structural types. On sequences of bits, therefore, `.*.` is just a local parallel-and operation.

To explore the EDSL components in more detail, consider the following exemplar definitions of `Entropy` instances for `Int32`, lists, and `Word8`:

```
instance Entropy Int32 where
  entropy _ = do {r <- randomM;
                  return (fromIntegral r)}
  x .+. y = x + y
  x .-. y = x - y
  x .*. y = x * y

instance Entropy a => Entropy [a] where
  entropy v = sequence [entropy u | u <- v]
  x .+. y = zipWith (.+.) x y
  x .-. y = zipWith (.-.) x y
  x .*. y = zipWith (.*.) x y

instance Entropy Word8 where
  entropy _ = do {r <- randomM;
                  return (fromIntegral r)}
  (.+.) x y = x `xor` y
  (.-.) x y = x `xor` y
  (.*.) x y = x .&. y
```

In the `Int32` declaration, the meanings of `.+.` etc. are just the usual arithmetic operators on 32-bit integers. In the list declaration, we lift the meanings of `.+.` etc. on the element types to act pointwise on the lists. Following the principle that types such as `Word8` be viewed as sequences of bits, the `.*.` operation on `Word8` is the bitwise *and*-operation (`.&.`). Our implementation also provides declarations for other types such as characters, and tuples[3].

2.2 Share Protocols

The operations of the `Entropy` class are all local operations that operate on local shares of values. We use these operations to construct operations which are semantically correct with respect to original shared value. These operations are the *share protocols*.

[2] When the type tells us everything about the size and shape of the argument, we don't need to refer to the value itself.

[3] It is not yet known how to share other kinds of values such as floating point numbers, or value-structure recursive structures (e.g. ordered trees), or functions etc. How to do so is an open problem.

```
add :: Entropy a => a -> a -> Protocol a
mul :: Entropy a => a -> a -> Protocol a
```

The multiply (`mul`) operation is global in that it involves interactions between the machines. The add operation (`add`) is a local operation. A richer language would also have other protocols, including methods for accepting shares of private inputs from users and for distributing shares of private results back to the users. For simplicity, in the latter cases we will simply configure these actions outside the EDSL.

Protocols such as `add` and `mul` are polymorphic over the `Entropy` class, so their precise action is structurally dependent on the types to which they are applied. To add together two private numbers which are represented by shares, we can simply add together the component shares (using `.+.`) and we are done. To multiply two private numbers, we have to compute nine partial products of their shares (Table 1). Each machine already has the

	y_A	y_B	y_C
x_A	$x_A y_A$	$x_A y_B$	$x_A y_C$
x_B	$x_B y_A$	$x_B y_B$	$x_B y_C$
x_C	$x_C y_A$	$x_C y_B$	$x_C y_C$

Table 1. Multiplication with Shares

values it needs to enable it to compute one of the entries on the diagonal. If each machine also communicates its x,y shares to its neighbor (according to the pattern in Figure 2), then every partial product in the matrix can be computed by somebody. For instance, machine B can compute $x_A y_B$, $x_B y_A$, and $x_B y_B$. We describe this in Haskell as follows:

```
mul :: Entropy a => a -> a -> Protocol a
mul x y = do
  (p,q) <- rotateRight (x,y)
  return ((x .*. y) .+. (p .*. y) .+. (x .*. q))
```

The low-level operation `rotateRight` transmits its argument `(x,y)` to its right hand neighbor machine, and receives a corresponding value `(p,q)` from its left hand neighbor. Recall, all three machines are operating loosely in lockstep, so all are executing the same instruction at around the same time. On receiving the neighbor's value, each machine computes the partial products, and returns the result to the calling procedure.

We need an additional refinement. If we performed multiple multiplications, we could end up rotating particular values to all three servers. This would then reveal enough information to reconstruct a private value, and so violate security. To avoid this, we take an extra step and re-randomize the shares before communication, as follows:

```
reshare :: Entropy a => a -> Protocol a
reshare s = do
   u <- entropy s
   v <- rotateRight u
   return (s .+. u .-. v)
```

Here each machine generates local entropy of the same size and shape as its argument, passes that entropy value to the right neighbor and receives a value from the left neighbor, and then calculates a new value for the share. As each random value is both added and subtracted from one of the shares, the overall (global) sum remains unchanged[4].

[4] We actually implement a standard optimization that removes the network communication implied here. Each machine generates its entropy using pseudo-random numbers. During initialization, we pass the random seed around to the next neighbor, so each machine can locally generate the entropy stream of its predecessor.

We can now write a secure version of the multiply protocol as follows:

```
mul :: Entropy a => a -> a -> Protocol a
mul x y = do
  (u,v) <- reshare (x,y)
  (p,q) <- rotateRight (u,v)
  return ((u .*. v) .+. (p .*. v) .+. (u .*. q))
```

Because each use of multiply communicates re-randomized shares, no information accumulates. Cryptographically, this makes the multiply operation *universally composable*, that is, we can use it repeatedly without fear of violating security.

Note, once again, that these operations are highly polymorphic. Because the `Entropy` class provides a structural extension over the overloading of the basic operations, the `add` and `mul` protocols are able to add and multiply whatever sizes and shapes of values we shall need (including lists and tuples).

3. Lookup Tables

Now that we have constructed the EDSL for share protocols, we can start writing programs. We quickly discover that we need data structures, and many standard assumptions no longer apply. In particular, we shall want table lookup (i.e. simple array indexing) but this becomes tricky—to say the least—when no individual server actually knows what index to look up!

A simple type for the lookup protocol is as follows:

```
indexing :: [a] -> Index -> Protocol a
```

where the `Index` type is some numeric type (e.g. `Word8`).

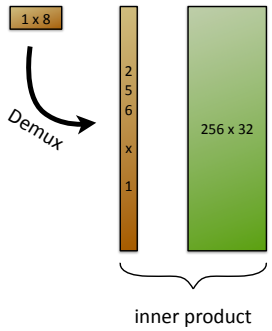

Figure 4. Inner Product with Demux

A moment's thought will convince us that the lookup algorithm has to act on all the entries of the table—otherwise we must have had *some* information as to what the index value was. Consequently, we should look to express the lookup protocol as some computation across the whole table. In fact, the form is very simple if we postulate a `demux` protocol that maps a binary representation of a value into a fixed-length, unary representation. Then the table lookup protocol is just a kind of inner product between the result of the demux protocol, and the table itself (see Figure 4). This intuition may be expressed more precisely as follows:

```
indexing :: [a] -> Index -> Protocol a
indexing table index = do
  ds <- demux index
  return (foldr1 (.+.) (zipWith mask ds table)
  where
    mask d entry = if d then entry else 0
```

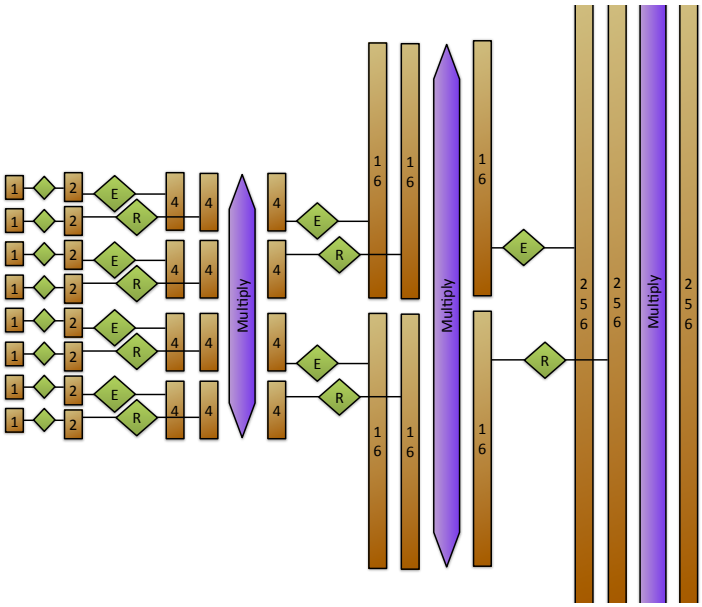

Figure 5. Parallel Manifestation of Demux Operation (for 8-bits)

A pure (i.e., non-share) demux function would map a binary representation of a value into a unary representation. For example, a 4-bit demux would take a 4-bit value and produce a 16-bit value (i.e. 2^4-bits) in which exactly one bit was set to 1, the other bits all being 0. We will adopt the convention that the demux bits are numbered from left to right. So, for example (using strings of 1's and 0's to represent lists of bits (Bools))[5],

```
demux "0000" = "1000 0000 0000 0000"
demux "0100" = "0000 0000 1000 0000"
demux "1111" = "0000 0000 0000 0001"
```

and so on. Note, we use big bit-endian representation for the binary bit-sequences, but count these unary bit-sequences from the left (consistent with Haskell practice elsewhere).

Correspondingly, a demux *protocol* would map a *share* of a 4-bit value, to a *share* of a 16-bit value. That is, if $x = x_A \oplus x_B \oplus x_C$, and if $d_i (i \in \{A, B, C\})$ is the result of running the demux protocol on the x_i, then $demux(x) = d_A \oplus d_B \oplus d_C$. For example, if we compute the demux of `0x8`, again going from 4-bits to 16-bits, then (one random outcome for) the d_i might be as follows:

```
d_A = "1011 0010 1110 1011"
d_B = "0011 0100 1100 1101"
d_C = "1000 0110 1010 0110"
```

Notice that only the 9th position (representing the value 8) has odd parity across all three shares; every other position has even parity.

Correctness of `indexing` is easy to establish. If we were dealing with pure functions (i.e. not share protocols), only one bit resulting from `demux` will be set, and this bit will select exactly the single row of the table corresponding to the original index. In the case of the share protocols, each d_i will contain a share of the true demux. That is, for each bit position j in the demux shares, $d_A(j) \oplus d_B(j) \oplus d_C(j) = 0$, except for the single bit position corresponding to the original index, in which case $d_A(j) \oplus d_B(j) \oplus d_C(j) = 1$. The `mask` function (written here as m) distributes across $\oplus$, so that $m(d_A(j) \oplus d_B(j) \oplus d_C(j), e) = m(d_A(j), e) \oplus m(d_B(j), e) \oplus m(d_C(j), e)$.

[5] Spaces are shown for ease of readability only.

3.1 Demux Protocol

All the computations involved in the definition of `index` were local (i.e. additions, or "scalar" multiplications where at least one value was known), except for those involved in the demux protocol. It is here that we shall have to work hard to minimize the number of global (multiply) operations.

As a pure function, demux can be expressed as a divide and conquer algorithm, satisfying the following equation.

```
demux (bs ++ cs)
   = [ b && c | b <- demux bs, c <- demux cs]
```

The list comprehension should be read as, "for each value `b` drawn from the list `bs`, and then for each value `c` drawn from the list `cs`, construct the value `b&c`." Thus, if `demux "10"` is given by `"0010"` and `demux "01"` is given by `"0100"`, then `demux 0b1001` is given by `"0000 0000 0100 0000"`.

Equivalently, we can express the cartesian product as a parallel multiply:

```
demux (bs ++ cs) = mul ds es
  where
    bs' = demux bs
    cs' = demux cs
    ds = expand (length cs') bs'
    es = replicate (length bs') cs'
```

where `expand n` duplicates each bit `n` times, and `replicate n` repeats the entire bit sequence `n` times. Thus, on the previous example,

```
expand 4 "0010"    = "0000 0000 1111 0000"
replicate 4 "0100" = "0100 0100 0100 0100"
```

Now a bitwise mul (`&`) produces `"0000 0000 0100 0000"`, i.e. 9.

We can do this fully in parallel as depicted in Figure 5. First we do a 1-bit demux (i.e. $0 \mapsto$ `"10"`, and $1 \mapsto$ `"01"`). Then we replicate and expand alternating 2-bit sequences to produce 4-bit sequences that are multiplied (i.e., anded) together. Similarly we replicate and expand 4-bit sequences into 16 bit sequences, and finally 16-bit sequences into a 256-bit sequence.

```
demux :: (Entropy word) => word -> Protocol [Bool]
demux w = demuxMerge (demuxBase (bits w))

demuxBase [] = []
demuxBase (b:bs) = [not b, b] : demuxBase bs

demuxMerge [bs] = return bs
demuxMerge bss  = do
    let (zss,odds) = chop2 bss
    let (xs,ys)    = unzip (map cartesian zss)
    zs <- mul xs ys
    demuxMerge (zs++odds)

expandReplicate (xs,ys)    = unzip [(a,b) | a <- ys, b <- xs]

chop2 []        = ([],[])
chop2 [x]       = ([],[x]) -- handles bit sizes that are not a power of 2
chop2 (x:y:xys) = case chop2 xys of (xs,ys) -> ((x,y):xs,ys)
```

Figure 6. Specification of Parallel Demux Operation (for any number of bits)

Figure 5 represents exactly the computation size and shape we need for AES. For reference, however, the code in Figure 6 provides a generic specification of the demux, even when the index word size is not a power of 2. Note, however, that the specification uses explicit lists of Booleans so, while it is fully executable, it is superficially inefficient. In practice, we re-implement the functions `demuxBase` and `demuxMerge` to work over packed representations of bits (e.g. 256 bits represented as a tuple of four 64-bit words). The definitions of these specific functions are provided in the Appendix.

The demux protocol on 8-bits requires just three multiplies (parallel-ands). But we can go one step further. The whole name of the game is to reduce the number of multiplications (because of their communication overhead), so we lift the whole definition of `demux` up to a list of indices, and return a list of results. Concretely, the version specialized to 8-bit indices is as follows:

```
demux :: [Word8] -> Protocol [Word256]
demux ws = do
    wss <- mul (map expnd4 ws) (map replt4 ws)
    yss <- mul (map expnd16 wss) (map replt16 wss)
    zss <- mul (map expnd256 yss) (map replt256 yss)
    return zss
```

Now `demux` has the same three calls to `mul` however long the input list of indices is. We can do this because the `mul` operation is overloaded on lists, and gathers together all the elements into a single packet for network communication. The corresponding type for `indexing` becomes:

```
indexing :: [a] -> [Word8] -> Protocol [a]
```

4. AES example

We are now ready to program AES in the share protocol EDSL. Figure 7 depicts the pattern of information flow, showing where information comes from and where it ends up. We assume that there is a machine which holds a plaintext version of a document (PT), and one that holds the key with which to encrypt the plain text. These machines send shares of their values (i.e. random values that—in this case—collectively xor to the originals) to the computation servers A, B and C. The servers then proceed to collaboratively encrypt the plain text—and this takes place without any individual server being able to discover anything about either the plain text or the key. Once the encryption computation is complete, the servers send their shares of the result to a machine responsible for collecting and constructing the cipher text (CT), which xors the individual shares to obtain the final result. Note that in some use cases the plain text, key, and cipher text may be on the same machine, and in others they may be on separate machines. The only security aspect we are aiming to maintain is that the computational servers A, B and C are distinct. Of course, as before, all communication is performed over SSL.

For the algorithm itself, we don't plan to give a detailed explanation of AES, partly because the main lookup protocol is so generic that we don't need to delve into the internals of the algorithm, and partly because the language aspects of what we are doing can be explained at a structural level.

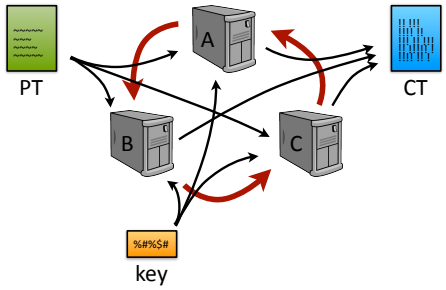

Figure 7. AES Configuration

The structure of the AES algorithm is shown in Figure 8. In the 128-bit version of AES, the plain text PT is a 16 byte chunk of the input, viewed as a 4x4 matrix of bytes. This matrix is called the *state*. In the *Key addition* phases, the bytes of the state are xor'd with key material generated from a pseudo-random generator that was seeded with the original crypto key. The *Shift row* phases rotate the individual rows of the state, and the *Mix column* phases does a column-based transformation based on modular polynomial multiplication. The *Byte substitution* phase replaces each byte in the state with a new byte, the value of which is defined by some Galois-field arithmetic, but whose software implementation is usually done by table lookup.

We decided to use a variant of the AES implementation that merges the byte substitution phase with the mix column phase, called *T-boxes*. The T-box formulation has become popular in both

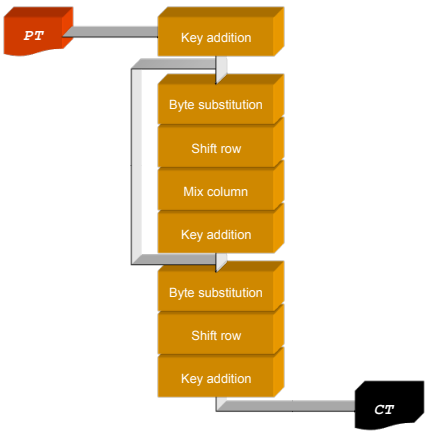

Figure 8. Structure of AES

software and hardware implementations [FM01] as it embodies all the algebraic complexity of AES in precomputed lookup tables. It was convenient for us because we had already produced an executable specification in Haskell for a different purpose. The core of the algorithm is given in Figure 9.

```
type State    = [[Word8]] -- 4x4
type RoundKey = [[Word8]] -- 4x4

encrypt :: (RoundKey,[RoundKey],RoundKey)
           -> [Word8] -> [Word8]
encrypt (key0,keys,keyN) inp = let
    input  = chop 4 inp
    first  = addRoundKey key0 input
    ending = seqMap cryptoRound keys first
    end    = finalRound keyN ending
  in  concat end

cryptoRound :: RoundKey -> State -> State
cryptoRound key u = let v = shiftRow u
                        w = map tFunc v
                    in addRoundKey key w

finalRound :: RoundKey -> State -> State
finalRound key u = let v = map (map (sbox !!)) u
                       w = shiftRow v
                   in addRoundKey key w

tFunc :: [Word8] -> [Word8]
tFunc [a,b,c,d] = let
    a' = tbox !! a
    b' = tbox !! b
    c' = tbox !! c
    d' = tbox !! d
  in (a' `xorPoly` rotLeft 3 b' `xorPoly`
      rotLeft 2 c' `xorPoly` rotLeft 1 d')
xorPoly = zipWith xor
```

Figure 9. T-Box Implementation of AES in Haskell

In this (inefficient) executable specification, the T-box is represented as a list of `Word32` with 256 elements. The T-box function `tFunc` looks up four byes in this table, and permutes and xors the result according to the T-box algorithm. The S-box table is simply a list of bytes (again specified in the definition of AES). The remaining functions such as `shiftRow` and `addRoundKey` are made up of simple rearrangements of structure and/or xor operations.

The control operator `seqMap` is a variant of `foldl` in which the arguments are flipped. For each of the round keys, `seqMap` applies the function `cryptoRound` to both the particular key in the list and the state, producing a new state for the next application. Think of `seqMap` as a kind of for-loop.

4.1 AES Share Protocol

It turns out to be simple to lift this specification into our EDSL for share protocols: the operation for table lookup is replaced by its monadic equivalent `indexing`, the `seqMap` control operator is replaced with the natural monadic equivalent, and the rest of the code is converted into monadic form in a standard manner (Figure 10).

```
encrypt :: (RoundKey,[RoundKey],RoundKey)
             -> [Word8] -> Protocol [Word8]
encrypt (key0,keys,keyN) inp = do
    let input = chop 4 inp
    let first = addRoundKey key0 input
    ending <- seqMapM cryptoRound keys first
    end    <- finalRound keyN ending
    return $ concat end

cryptoRound :: RoundKey -> State -> Protocol State
cryptoRound key u = do
    let v = shiftRow u
    w <- mapM tFunc v
    addRoundKey key w

finalRound :: RoundKey -> State -> Protocol State
finalRound key u = do
    v <- mapM (mapM (indexing sbox)) u
    let w = shiftRow v
    addRoundKey key w

tFunc :: [Word8] -> Protocol [Word8]
tFunc [a,b,c,d] = do
  a' <- indexing tbox a
  b' <- indexing tbox b
  c' <- indexing tbox c
  d' <- indexing tbox d
  return (a' `xorPoly` rotLeft 3 b' `xorPoly`
          rotLeft 2 c' `xorPoly` rotLeft 1 d')
```

Figure 10. AES as a share protocol

Already this version takes advantage of the bit-level parallelism built into the `indexing` function—within `tFunc`, four bytes of the state are looked up in parallel. However, there are four sets of the 4-byte lookups, and they are indexed sequentially. This means that this implementation of AES requires 12 (networked) multiplications (i.e. parallel-ands) per round (i.e. 4 indexing operations, each requiring 3 parallel-ands).

We can improve performance by modifying the definition of `cryptoRound` so that it does a SIMD lookup operation on all the bytes of the state at once.

```
tFunc :: [Word8] -> Protocol [Word8]
tFunc [a,b,c,d] = do
  [a',b',c',d'] <- fmap (chop 4) $
                   indexing tbox (concat [a,b,c,d])
  return (a' `xorPoly` rotLeft 3 b' `xorPoly`
          rotLeft 2 c' `xorPoly` rotLeft 1 d')
```

Now, all the indexing operations are performed together, so we have just 3 (bit-parallel) multiplies per round. More bits are being being anded each time, but the overhead of enlarging the vector of values in a single parallel-and is very small.

It would be a pity if we always had to do these kinds of performance improvements by hand. In the next section we show how we use compilation techniques to transform the direct SMC version into a more efficient implementation.

5. Automated Performance Improvements

There are a number of improvements that would lead to a more efficient SMC implementation. In particular, we would like to eliminate the list manipulations and—whenever possible—to replace sequential multiplications by grouping many together in one larger parallel SIMD-style operation. This section describes an automated approach to doing so, which may be generalized to other EDSLs. It is fairly technical and can be skipped on first reading.

We start by re-interpreting the algorithm in an environment where some of the types and operations have been replaced with *symbolic* counterparts. For example, the usual type for bytes, `Word8` is replaced by the following symbolic version (using Haskell's GADT notation for consistency later on):

```
data Word8 where
  Var8    :: Name -> Word8
  SBox    :: Word256 -> Word8
  Xor8    :: Word8 -> Word8 -> Word8
  GetByte :: Int -> Word32 -> Word8
```

The constructor `Var8` represents a completely unknown value, while the other constructors are for values constructed with the corresponding functions, thus the symbolic version of a function is simply the constructor:

```
xor = Xor8
```

Executing the algorithm in this environment essentially performs symbolic computation: types that were left concrete are evaluated away, and the final result is a symbolic description of the algorithm.

For the AES example, we left functions and lists as concrete types, while we used symbolic representations for the word types and the `Protocol` monad. Executing the `encrypt` function in this environment performs a partial evaluation, producing a residual version of the algorithm where all the loops are unrolled and there are no list manipulations. We perform additional optimizations on this data structure to derive the final efficient algorithm, which we describe later.

To represent symbolic `Protocol` computations we used a *syntactic monad* with all binds normalized to the right, so the datatype resembles a list of (appropriately typed) primitive operations, `ProtocolOp`, terminated by a `Return`:

```
type Protocol = SMonad (->)

data SMonad :: (* -> * -> *) -> * -> * where
  Return :: a -> SMonad fun a
  (:>>=) :: ProtocolOp a
         -> a `fun` SMonad fun b
         -> SMonad fun b

data ProtocolOp :: * -> * where
  Mul     :: Entropy a => a -> a -> ProtocolOp a
  LkpTBox :: Word256 -> ProtocolOp Word32
```

The type is parameterized by the function-space constructor so that we can use it with both concrete and symbolic function-spaces: the first one is useful for symbolic evaluation, while the second one is useful for analyzing `Protocol` computations. As in the case for pure symbolic evaluation, defining the symbolic functions is fairly simple, except that in this case we have to make sure that they are in the normal form that we chose:

```
instance Monad Protocol where
  return a          = Return a
  Return a    >>= f = f a
  (op :>>= k) >>= f = op :>>= \r -> k r >>= f

mul :: Entropy a => a -> a -> Protocol a
mul x y = Mul x y :>>= return
```

The final step before we have a representation of the algorithm that can be analyzed and optimized is to write a function that will replace the concrete functions in a monadic computation with their symbolic equivalents:

```
type Code     = SMonad (:->)
data a :-> b  = a :-> b

compile :: Supply Name -> Protocol a -> Code a
compile _     (Return a)  = Return a
compile names (op :>>= k) =
  let (n1,n2) = split2 names
      x       = case op of
                  Mul {}     -> newVar n1
                  LkpTBox {} -> newVar n1
  in op :>>= x :-> compile n2 (k x)
```

The basic idea behind `compile` is to replace a function's argument with a symbolic value, which is then used to evaluate the function's body. Symbolic values are generated by the overloaded function `newVar` (the actual name generation uses the technique described in [Aug94], thus avoiding the need to plumb the name supply around). Note the intricate interaction between generalized algebraic datatypes and overloading in the definition of the symbolic value, `x`: pattern matching on the operation reveals the type of the expected result which, in turn, allows the compiler to resolve the overloading of `newVar`[6].

At this stage, we have a completely symbolic (but still typed) representation of the algorithm, which is suitable for analysis and rewriting. Our goal is to group single multiplications into multiple-value SIMD-style multiplications. Note that the operations of the `Protocol` monad *commute* in the sense that they can be rearranged freely, only subject to data dependencies: as long as the three computational nodes operate in sync, it does not matter in what order they compute the multiplications. Using this fact, we can write a code transformation that rearranges the code by data-dependencies: first we execute instructions that only depend on the function arguments, next we execute instructions that depend on function arguments and the results of the first group, and so on. The details are presented in function `rearrange`:

```
rearrange :: [Name] -> Code a -> Code a
rearrange _ (Return x) = Return x
rearrange us p =
  let (vs, m1, m2) = pullUp us p
  in m1 (rearrange vs m2)

pullUp :: [Name] -> Code a
       -> ([Name], Code b -> Code b, Code a)
pullUp us (Return a) = (us, id, Return a)
pullUp us (op :>>= x :-> m)
  | all (`elem` us) (fvs op)
     = ( newDefs ++ vs
```

[6] In the case of multiplication, the overloading is resolved because the symbolic version of the Entropy class supports generating symbolic values.

```
      , \k -> op :>>= x :-> this k
      , next )
  | otherwise = (vs, this, op :>>= x :-> next)
  where
  (vs,this,next) = pullUp us m
  newDefs        = case op of
                     Mul {} -> fvs x
                     LkpTBox {} -> fvs x
```

Most of the work for this transformation is performed by the function `pullUp` which, given a set of names and a computation, splits the computation into two parts: those instructions that only depend on the names (represented in continuation-passing style) and all other instructions. While performing the analysis, `pullUp` also computes the new set of defined names, to be used in the following iteration of the analysis.

Having rearranged the instructions by data-dependency, all we need to do is implement another code transformation that identifies independent adjacent multiplications and combines them into a single SIMD-style multiplication. For illustration purposes the code transformation that we show uses nested (symbolic) pairs to perform multiplications in parallel. In a similar fashion we could write another code transformation that flattens the nested pairs into a more efficient data structure (e.g., an array of a fixed size):

```
joinMul :: Code a -> Code a
joinMul (Mul a b :>>= c :-> (Mul x y :>>= z :-> m))
  | all (`notElem` fvs c) (fvs x ++ fvs y)
    = joinMul
    $ Mul (Pair a x) (Pair b y) :>>= Pair c z :-> m
joinMul (op :>>= c :-> m) = op :>>= c :-> joinMul m
joinMul m = m
```

At this point we have transformed the original (somewhat inefficient) algorithm into one with a better structure for run-time performance. The approach of deriving the efficient implementation in this manner gives us some confidence in the correctness of the final efficient algorithm because (i) we used a typed representation throughout the development, and (ii) the transformation steps are small and independent, so we can can examine and evaluate them in isolation. Furthermore, while we used AES as an example, the overall idea is quite general, and can be used for other algorithms also.

Throughout this section we described how to derive an efficient algorithm for encrypting a single AES block. However, a typical use of AES involves processing many blocks, which gives rise to yet another opportunity for speeding the implementation: we can improve the performance by processing multiple AES blocks in parallel. This works well in certain encryption modes—in Galois counter mode (GCM), for example, where AES is used from a fixed starting point to produce a kind of one-time pad. Pipelining is not appropriate in other modes: in cipher-block chaining (CBC), for example, the result of one encryption is xor'd with the next block prior to its encryption, so pipelining is not possible.

When we can do multiple blocks at the same time, we can either use concurrency primitives to execute multiple encryption functions at once, and/or generate custom encryption functions that encrypt multiple blocks at once. We actually do both for different reasons, as we explain in the next section.

The technique described in this section make it extremely easy to derive such multi-block encrypting functions. For example, to generate a function that encrypts 4 blocks at once, we simply run our "compiler" on the following code:

```
keys :-> b1 :-> b2 :-> b3 :-> b4 :->
  do r1 <- encrypt keys b1
     r2 <- encrypt keys b2
     r3 <- encrypt keys b3
     r4 <- encrypt keys b4
     return (Pair4 r1 r2 r3 r4)
```

Evaluating this program symbolically unrolls all loops and groups together all multiplications. Note that because the blocks are processed independent of each other, all their multiplications end up grouped together into large SIMD-style multiplications, which helps performance.

6. Performance

The first implementation we did was extremely inefficient. Our initial focus was on the correctness of the specification of the table-lookup algorithm and avoided making any (possibly premature) commitments to particular optimizations. Consequently, the first implementations were slow, executing AES at about one block per second. We then systematically identified performance hot spots and bottlenecks, and corrected them. All of these mitigations were quite localized, so the code we ended up with is still very recognizably similar to the original specification.

The major optimization was mentioned above—to perform all 16 indexing operations from a single AES state in one go, exploiting the fact that we were able to build the `indexing` function with a parallel SIMD-style capability.

Laziness was both good and bad. We make extensive use of laziness in control structures built from (virtual) data structures, but at the same time we have to control laziness in the results of Protocol computations. Laziness can lead to large intermediate computations being created in the heap before consumers for the values eventually force their evaluation. In the AES example, no control decisions are ever taken on the basis of shares of private values—naturally, given their random nature—so the only evaluation-demand came from serializing intermediate values for network sends. Judicious use of `seq` and its more vigorous cousins (e.g. reduction to full normal form by `NFData`) handled this potential problem very easily.

As mentioned previously, the first version of `indexing` used lists of booleans to represent the result of `demux`. Once we were confident that the definition was correct, they weren't particularly hard to replace with packed representations. The calculations of the precise bit-manipulations are tedious however, so we have included the code in the Appendix.

We had some issues with GHC not inlining the primitive monad operators (i.e. `>>=`). This created a significant overhead by itself, and additionally prevented GHC from carrying out other optimizations. We traced the problem to our overloading the randomization operations to work in multiple monads (we needed `IO` and `Protocol`). Once we separated these cases, the GHC inliner was able to do its magic.

After calculating the `demux` of the index, the `indexing` function computes a masked xor of the whole table. This is such a central operation, and repeated so often that we had to ensure it was maximally efficient. We could probably have coaxed Haskell to produce optimal code, but for this simple operation it was easier to just write it in C and call the C function through Haskell's foreign function interface (FFI)—the FFI is to Haskell as `asm` is to C. On similar lines, we also introduced a new interface to the SSL Haskell bindings that allowed us to build `Storable` values directly rather than build byte strings simply to have them converted into storables.

We also upgraded the standard Haskell random number generator to be cryptographically random as well as statistically random. We used the secure RNG design based on AES, as provided by NIST [BK12]. Happily, this did not run any slower, and on machines with special AES instructions runs significantly faster than the standard randoms in Haskell.

All of these optimizations improved both space and time usage of the non-networked part of the computation. Earlier versions of the code had heap residency of 8-10MB plus a couple of space leaks. After the optimizations described here, the heap residency was stable at around 220KB. We considered that sufficiently optimized for our purposes.

6.1 Latency

With heap requirements being well-managed, the limiting factors on performance become CPU, network bandwidth, and network latency. Given that multiple network communications take place in the midst of any non-trivial share computation, the general pattern of any computation is shown in Figure 11.

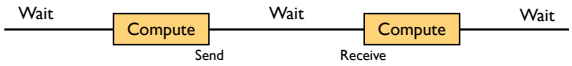

Figure 11. Effect of Network Latency

We do some work, then perform a network send to communicate values, and then wait to receive the values to come from the neighbor machine. What does this mean for AES? There are 10 rounds in the AES algorithm, each round requiring one (parallel) table lookup. Each table lookup requires three (global) parallel-and operations, and each of these parallel-and operations requires a round-robin communication between the servers. On our benchmarking setup[7], the latency of a 1kbyte SSL communication between machines is around 300 μs. Encrypting a single block, therefore, will cost 9 ms just in network latency. Indeed, we measure that computing one block of AES in isolation takes around 14.3 ms. This number is important: as we increase the opportunities for parallelism we will decrease the overall time per block, but we will never decrease the block latency below this figure. Indeed, it may increase slightly.

When encrypting a single block at a time, the network latency in every table lookup is the limiting factor—even on a fast network—as CPU loads are low, and network bandwidth is low. As described in the previous section, there are two kinds of parallelism we can introduce to reduce the impact of this network latency.

First, we can process many blocks in parallel in a single instance of the protocol. Doing two or more blocks at once increases the amount of work we do before incurring the network latency, and it does not increase the wait phase significantly (though we may start to increase the number of packets sent per operation). In effect, this form of SIMD parallelism shares the cost of network latency amongst multiple blocks, thus reducing the latency overhead on any individual block. Secondly, we can run multiple instances of the protocol in parallel, each with their own SSL connection. This parallelism is simply traditional concurrency, where concurrent execution allows the wait time in one thread to be shadowed by the compute time in another thread.

The results of varying these parameters is shown in Figure 12. Each point is the average of 10 runs. In both charts, the Y-axis is amortized encryption time (in ms per 128-bit AES block). In the upper chart, the X-axis specifies the number of blocks being processed by any single compute phase, and the different curves represent the number of concurrent threads independently processing groups of blocks. In the lower chart, the roles of these are transposed.

[7] All the benchmarks were carried out on three physically separate machines networked with gigabit ethernet. The systems are identical, running quad-core 3.1 GHz Intel Nehalem-C CPU with 16GB of memory and accelerated AES instructions (for random number generation). To ease benchmark administration, each worker runs CentOS 6.2 as a KVM virtual machine.

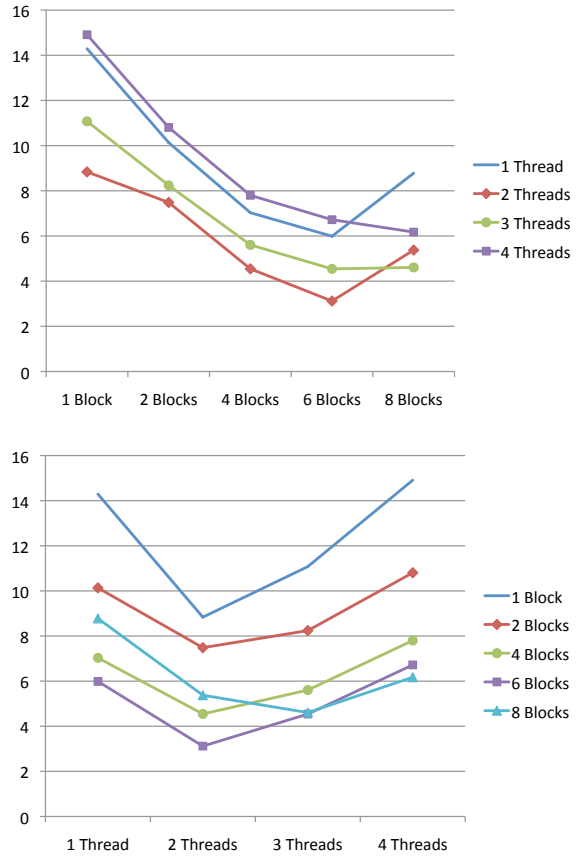

Figure 12. Time (in ms) to encrypt an AES Block

In these specific experiments, the winner is clear. Running two concurrent threads, each of which is processing 6-AES blocks in SIMD style, produces the minimum amortized time per block of 3.1 ms. This translates to 320 AES blocks per second, or a secure-encryption throughput of 41 kilobits per second.

In the lower chart, all the curves rise after two or three threads. The local network is sufficiently fast that the latency does not need many threads to become fully shadowed, and having additional threads simply introduces contention. In a different network setting, such as connecting three servers owned by different organizations across the public internet, we expect that many more threads will be required to shadow the much larger network latency that would arise.

Network bandwidth was not a limiting factor for us. In the fastest case (6 blocks, 2 threads) packet sizes ranged from 166 to 4412 bytes, with an average packet size of 784 bytes. Packets flowed at a rate of 3195 per second, translating to an average network bandwidth of 20 megabits per second. Likewise, CPU utilization was not limiting (given our 4 core machines). In the 1 block, 1 thread case, CPU utilization was 0.75. It rose to 1.45 in the 6 block, 2 threads case, and to 1.70 in the 6 blocks, 4 threads case.

7. Related Work

After the seminal work of Yao [Y86], the FairPlay system was arguably the first major leap towards practicability [MNPS04]. The Fairplay system provided a full compilation pathway from a version of C down through compilation that carried out various optimizations of the underlying Yao *garbled circuit* scheme. FairPlay had

array lookup and assignment (like our table lookup) and the Fairplay compiler builds multiplexing and demultiplexing circuits as explicit boolean circuits, though there is no effort to maximize parallelism as we do here. In fact, it is an open question as to whether a SIMD-style parallelism is possible in the Yao setting.

There have been a number of implementations of AES in FairPlay and other Yao-style implementations that also assume the same honest-but-curious security model. In two-party implementations of AES, one party typically knows the key, and the other part knows the cypher text. Damgaard and Keller attempted to gain a fast AES by exploiting the algebraic structure of the algorithm itself [DK10]. They proposed several variants, the best of which required (approximately) an average of 2200 elementary operations in 70 rounds to encrypt one AES block. Their implementation with three participants in a local network showed that one block can be encrypted in 2 seconds, though they argue that this result could be improved by optimizing the implementation.

Huang *et al*'s recent USENIX paper focused on how to optimize Yao garbling to make the protocols significantly faster than before [HEKM11], in many cases faster than corresponding custom-designed protocols. In their setting—as with ours—the key-schedule is expanded outside of the crypto framework (for them, because one party knows the key entirely; for us, because the key is known by the initiator of the computation), so in neither case does key expansion contribute to the cost of the computation. The AES implementation is done on the original formulation of AES, where the mix-column is performed explicitly. This is done just using XOR's, which come pretty much for free in their framework. Large circuits still get generated, but they are produced in a pipelined fashion so never need to be stored all in one place. When table lookups are performed, the tables are viewed as circuits themselves, requiring garbled version of the tables to be generated and transmitted. AES encryption time is 0.06 seconds per block (after preprocessing). That is, about 17 blocks per second.

Pinkas et al focus on a more stringent security model, where the protocol still succeeds despite a certain proportion of malicious players who attempt to corrupt the protocol [PSSW09]. In this setting, again based on Yao garbling, a single block AES encrypt takes 8 seconds under honest-but-curious assumptions (not counting preprocessing time), and 661 seconds under malicious adversary assumptions.

The Sharemind system [BLW08] is built on the same principles as the system described here. It too has three servers, and performs arithmetic sharing. In some dimensions, the Sharemind system is more fully engineered than our EDSL, in that it comprises a stand alone input language SecreC (i.e. much of C, along with annotations for secrecy), a compiler, a low-level virtual machine interpreter, and theorem proving support for privacy proofs. On the other hand, the fact that we built an EDSL on Haskell means that we are able to bypass most of those components and inherit them from the host language directly. Consequently, we have been able to implement shares of private values for many new families of data types, and rapidly explore many different instantiations of higher level protocols. The Sharemind system has produced many advances in protocols, notably including division, and has been used for a number of real world application demonstrations. In the last couple of months, the Sharemind developers have begun work on their own version of AES.

Bain et al also constructed a language for privacy computation as an embedded DSL in Haskell [BMS+11]. Their language is agnostic about the underlying cryptographic computational substrate, and were able to hook into multiple computational backends (Shamir-sharing, Yao garbling, and FHE). Their main focus was on establishing privacy proofs rather than on algorithmic improvements, and did not attempt to optimize their implementations.

A significant breakthrough on the underlying security model has come from Gentry's *fully homomorphic encryption* (FHE) scheme [G09]. FHE doesn't rely on multiple parties, but does all the computation within crypto mathematics. In Gentry's scheme, individual bits are randomly encoded as large noisy vectors, i.e. points in a space that are close (but not too close) to the points on a regular multi-dimensional lattice. Operations of addition and multiplication on the lattice also perform corresponding calculation on the cipher texts, the results of which are revealed once the resultant cipher texts are decrypted. There is a detail to manage though: as these lattice operations are performed, the *noise* increases (i.e. the cipher texts move further and further away from the lattice points), requiring use of a *recrypt* operation to reset the noise to something like original levels. FHE schemes are currently *very* inefficient, with initial implementations requiring about two seconds to compute a single *and*-operation [GH11]. However, progress in this area is rapid, and AES may be within sight of FHE implementations relatively soon.

8. Conclusion

The programming language community has an excellent understanding of the performance profile of standard CPU RAM architectures, even though multi-cores and cache and other memory hierarchies make many things very much more complicated than they used to be. SMC-style computation on private values presents a very different execution model, and one that is only beginning to receive attention.

In all existing manifestations of computation on private values, multiplication (both arithmetic and boolean) is exceedingly expensive compared with every other operation. In arithmetic sharing (the setting of this paper) the expense comes from the entropy and network accesses required. In Yao garbling, the expense arises because multiplications create significant expansions in the size of the circuit that has to be communicated and evaluated. In fully homomorphic encryption, the expense comes from multiplications dramatically increasing the noise within the crypto value. These force the programmer to trade off between using larger security parameters or requiring more frequent reset operations, each of which entails using a homomorphic instance of the decrypt operation.

When optimizing computations in SMC or FHE computational models, we need to approach multiplications with the same mindset we use for disk accesses—how do we minimize them, block them together, and hide the latencies they incur? Some of these performance-improving techniques can be implemented within the secure computation technique itself—for example, some of the SMC and FHE approaches are moving to produce SIMD versions of the basic multiply operation (e.g. [SF11])—but that only goes so far. The rest of the optimizations have to come from programming and/or compilation techniques that are designed to optimize for this strange execution model.

This paper contains an example of the kind of algorithmic rethinking that is required. We gained well over an order of magnitude for AES based mostly on a good algorithm for table lookup and judicious use of parallelism. The field is in its infancy regarding how we do other kinds of algorithms, and progress here can have a dramatic impact.

Acknowledgments

Mark Tullsen helped with network and testing setup, and Sally Browning aided in generating benchmark figures. Abhi Shelat and John Mitchell provided helpful discussions about the security of the demux. Dan Bogdanov and Jan Willemson helped with understanding the share protocols used in this implementation.

References

[NIST01] National Institute of Standards and Technology (NIST). *FIPS 197: Advanced encryption standard, 2001*. Available at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[Aug94] Lennart Augustsson, Mikael Rittri, and Dan Synek *On Generating Unique Names*. Journal of Functional Programming (1994), 4 : pp 117-123

[BLW08] D. Bogdanov, S. Laur, J. Willemson. *Sharemind: a framework for fast privacy-preserving computations*. In Proceedings of 13th European Symposium on Research in Computer Security, ESORICS 2008, LNCS, vol. 5283. Springer-Verlag, 2008.

[BCD+08] P. Bogetoft, D. L. Christensen, I. Damgaard, M. Geisler, T. Jakobsen, M. Kroeigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft, *Secure Multiparty Computation Goes Live*. Financial Cryptography and Data Security, R. Dingledine and P. Golle (eds), LNCS Vol. 5628, Springer-Verlag 2009.

[BK12] E. Barker and J. Kelsey. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. NIST Special Publication 800-90A. http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf

[BMS+11] A.M. Bain, J.C. Mitchell, R. Sharma, D. Stefan and J. Zimmerman. *A domain-specific language for computing on encrypted data*. Invited Talk. In Foundations of Software Technology and Theoretical Computer Science, 2011.

[DK10] I. Damgaard and M. Keller, *Secure Multiparty AES*, Financial Cryptography and Data Security, R. Sion (ed), LNCS Vol. 6052, Springer-Verlag 2010.

[FM01] Two Methods of Rijndael Implementation in Reconfigurable Hardware, *V. Fischer and M. Drutarovsk*. Cryptographic Hardware and Embedded Systems (CHES 2001). LNCS Vol. 2162. Spring-Verlag 2001.

[G09] C. Gentry, *Fully homomorphic encryption using ideal lattices*. ACM Symposium on Theory of Computing (STOC 2009), 2009.

[GH11] C. Gentry and S. Halevi. *Implementing Gentrys fully-homomorphic encryption scheme*. Advances in Cryptology (Eurocrypt 2011), LNCS Voli. 6632, Springer-Verlag, 2011.

[G04] O. Goldreich, *Foundations of Cryptography, Vol 2: Basic Applications*. Cambridge University Press, 2004.

[HEKM11] Y. Huang. D. Evans, J. Katz, L. Malka, *Faster Secure Two-Party Computation Using Garbled Circuits*. In 20th USENIX Security Symposium, San Francisco, 2011.

[MNPS04] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, *FairPlay – a secure two party computation system*. Proceedings of the 13th conference on USENIX Security Symposium - Vol. 13, 2004.

[PSSW09] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. *Secure Two-Party Computation is Practical*. Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology. ASIACRYPT 09. Springer-Verlag 2009.

[SF11] N. P. Smart and F. Vercauteren. *Fully homomorphic SIMD operations*. Manuscript at http://eprint.iacr.org/2011/133, 2011.

[Y86] A. C. Yao. *How to generate and exchange secrets*. In Proceedings of the 27th IEEE Symposium on Foundations of Computer Science, 1986.

A. Appendix

Here is a listing of the functions called by `demux`. For performance, is it critical to use packed representations of the bit sequences, and while it is not trivial to work out the bit manipulations involved, they can all be derived by calculation from the general specification provided earlier.

The `expnd4` and `replt4` functions map from the original 8-bit word in Figure 5 through to the 16-bit words prior to the first multiplication. As the diagram indicates, `expnd4` operates on the odd numbered bits, and `replt4` on the even numbered bits (counting 0-7). In both functions, initially a 0 maps to 01, and a 1 maps to 10 (representing a 1-bit demux). Then, in `expnd4` these sequences are expanded bitwise, and in `replt4` they are replicated.

```
expnd4 :: Word8 -> Word16
expnd4 b  = case 8 * sel 7 b + 4 * sel 5 b
                 + 2 * sel 3 b + sel 1 b of
            0  -> 0x3333
            1  -> 0x333c
            2  -> 0x33c3
          -- etc
            15 -> 0xcccc

replt4 :: Word8 -> Word16
replt4 b = case 8 * sel 6 b + 4 * sel 4 b
               +  2 * sel 2 b + sel 0 b of
            0  -> 0x5555
            1  -> 0x555a
            2  -> 0x55a5
          -- etc
            15 -> 0xaaaa

sel :: Int -> Word8 -> Word8
sel k b = shiftR b k .&. 0x01  -- select bit k
```

In `expnd16` and `replt16` the argument represents four 4-bit words: `expnd16` operates on the odd numbered nibbles, and `replt16` on the even numbered ones (counting 0-3). In `expnd256` and `replt256` the argument represents two 16-bit words. The resulting 256-bit words are represented as four 64-bit words.

```
expnd16 :: Word16 -> Word32
expnd16 w  =  expn ((w .&. 0x00f0) `shiftR` 4) .|.
 ((expn ((w .&. 0xf000) `shiftR` 12)) `shiftL` 16)
  where  expn w4 = case w4 of
                     0  -> 0x0000
                     1  -> 0x000f
                     2  -> 0x00f0
                -- etc
                     15 -> 0xffff

replt16 :: Word16 -> Word32
replt16 w = fromIntegral ((w .&. 0x000f) * 0x1111)
 .|. (fromIntegral (((w .&. 0x0f00) `shiftR` 8)
        * 0x1111) `shiftL` 16)

expnd256 :: Word32 -> Word256
expnd256 w
  = W256 (scale 12) (scale 8) (scale 4) (scale 0)
 where
  scale :: Int -> Word64
  scale off = scale1 off 3 .|. scale1 off 2
              .|. scale1 off 1 .|. scale1 off 0
  scale1 :: Int -> Int -> Word64
  scale1 off ix = (if x `testBit` (off + ix)
                     then 0xFFFF else 0x0000)
                             `shiftL` (ix * 16)
  x :: Word32
  x = w `shiftR` 16

replt256 :: Word32 -> Word256
replt256 w = W256 w64 w64 w64 w64
  where
    w64 = (x `shiftL` 48) .|. (x `shiftL` 32)
          .|. (x `shiftL` 16) .|. x
    x :: Word64
    x = fromIntegral (w .&. 0xffff)
```