Application-Scale Secure Multiparty Computation

John Launchbury, Dave Archer, Thomas DuBuisson, and Eric Mertens

Galois, Inc*

Abstract. Secure multiparty computation (MPC) permits a collection of parties to compute a collaborative result without any of the parties or compute servers gaining any knowledge about the inputs provided by other parties, except what can be determined from the output of the computation. In the form of MPC known as linear (or additive) sharing, computation proceeds on data that appears entirely random. Operations such as addition or logical-XOR can be performed purely locally, but operations such as multiplication or logical-AND require a network communication between the parties. Consequently, the computational overhead of MPC is large, and the cost is still measured in orders of magnitude slowdown with respect to computing in the clear. However, efficiency improvements over the last few years have shifted the potential applicability of MPC from just micro benchmarks to user-level applications.

To assess how close MPC is to real world use we implement and assess two very different MPC-based applications—secure email filtering and secure teleconference VoIP. Because the computation cost model is very different from traditional machines, the implementations required a significantly different set of algorithmic and compiler techniques. We describe a collection of the techniques we found to be important, including SAT-based circuit optimization and an optimized table lookup primitive.

1 Introduction

It is scarcely possible to read the news without seeing yet another reason to be able to perform computation on encrypted data. The cryptography community has long known that some kinds of computations on encrypted data are possible—at least in principle. This was notably demonstrated by Yao's seminal work on secure multiparty computation [Y86], and most radically by Gentry's work on fully homomorphic encryption (FHE) [G09]. While FHE is very new and still far from practical, there has been significant effort in the last few years to make MPC usable in practice.

MPC 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 (other than what is derivable from the final result of the

* 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.

Z. Shao (Ed.): ESOP 2014, LNCS 8410, pp. 8–26, 2014.

computation). In recent years, the variant of MPC called *linear shared computation* has been producing significant performance wins [BLW08, LAD12, DKL+13].

When we say "performance wins", we should put it in context: on test cases such as securely decrypting AES-encrypted text, we have been seeing linear sharing achieving execution times of around 3–30ms per 128-bit block, which corresponds to a slowdown of around four to five orders of magnitude compared with computation in the clear. Significant though this slowdown is, it compares well with Yao and especially with FHE, whose current slowdowns appear to be respectively around six and nine orders of magnitude in our experience.

There are two fundamental reasons why secure computation proceeds more slowly than computation in the clear. First, all secure computations have to be performed generically across all possible input and internal values (otherwise information is revealed), though there are neat algorithms which can sometimes amortize this somewhat across multiple accesses. Second, the multi-party schemes (both Yao and linear sharing) require significant network communication, typically growing linearly with the size of the function being evaluated.

MPC 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 secure computation 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 snooping 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.

1.1 Contributions of This Paper

In this paper, we address the challenge of scaling secure computation to a level required by applications. We implement two: a mail filter, which matches encrypted email texts against regular expressions, and VoIP teleconference calling, which merges and clips multiple audio streams in real-time.

To implement these, we used the *ShareMonad*, a Haskell-embedded domain-specific language for programming secure multiparty computations, with a linear-sharing backend [LAD12]. The ShareMonad view considers the secure multiparty computational substrate as an *MPC-machine*—an abstract machine with highly non-standard interface and performance properties. The implementation comes with a variety of ad-hoc techniques for minimizing expensive operations, either by reducing the overhead of individual operations (through exploiting opportunities for SIMD-like parallelization), or by hiding residual latencies involved in network-based operations. To scale to the size and performance required by our target applications, we further developed the backend optimizations. In particular:

- We developed and implemented many compile-time optimizations, including SAT-based reasoning to replace (expensive) AND-operations with (cheap) XOR-operations, and balancing and packing of global operations to minimize the number and size of network communications.
- We also created a new version of the table lookup primitive, introduced in [LAD12]. This performs secret lookup of an n-bit index in a public table using $log(n)$ global operations (as before), but where each global operation now communicates no more than $2^{1+n/2}$ individual bits. We also optimize the local computations involved in the table-lookup with some pre-computation on the table. Together, these make a huge difference in both computation and network performance. In effect, the compiler uses the table lookup protocol as a mechanism for building custom wide-word instructions that are generated based on the program.

2 Background

The secure computation scheme we use is simple linear (arithmetic) 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 1. Some more complex protocols require less uniform computation and communication patterns, but we won't need them here.

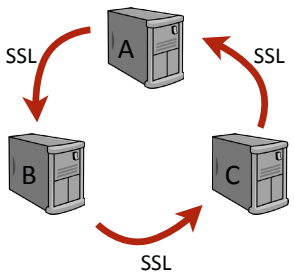

Fig. 1. Machine Configuration

The diagram shows the links protected with SSL. The critical point is that the links are protected by some mechanism, otherwise a network snooper could access the three random shares of a value and so reconstruct the original. For performance and thread-safe reasons, we use a homegrown commsec package instead of OpenSSL, that is 3× faster on small messages.

In an *arithmetic* 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 shares are produced originally 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, each one of the 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[1]. Subsequently, the computational protocols maintain the 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 of a sequence $[x, y, z]$ will be shared as three (equal length) sequences of 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 the Share-Monad library) as if they were sequences of individual bits (i.e. elements of `Int1`, where multiplication is just boolean 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).

To add together two private numbers which are represented by shares, we can simply add together the component shares and we are done. To multiply two private numbers, we have to compute nine partial products of their shares (Fig. 2).

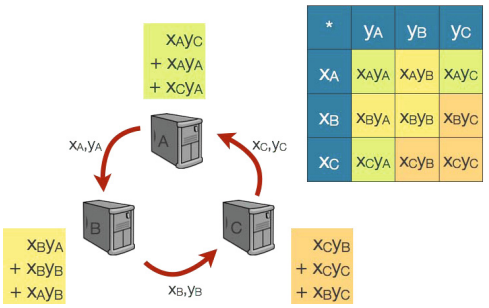

Fig. 2. Computing the Partial Products

Each machine already has the values it needs to enable it to compute one of the entries on the diagonal. If each machine also communicates its shares of x and y to its neighbor (according to the pattern in Fig. 1), then every partial product in the matrix can be computed by somebody. All three machines are

[1] Even if given two of the values, x_A and x_C say, every possible value for x has equal probability, depending entirely on the value of x_B.

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 three partial products, XORs them together, and now has a share of the full product.

We need an additional refinement. If we performed multiple multiplications in a sequence, we could easily end up rotating particular share 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. Because of this, each use of multiply communicates re-randomized shares, and so no information accumulates. Cryptographically, this makes the multiply operation *universally composable*, that is, we can use it repeatedly without fear of violating security. As the addition operation requires no communication, it automatically has this property.

3 Applications

We selected two target applications: a secure mail filter, and secure VoIP teleconference calling. They exhibit a significant divergence in application characteristics. The mail server is a batch process that evaluates regular expressions, and the VoIP system is a soft real-time system using simple audio algorithms. We describe each of the applications, including their set-up, and then turn to consider how to scale the secure computation components in each.

3.1 Secure Mail Filter

In the secure mail filter architecture in Fig. 3, the sender S writes an email in Thunderbird. We created a plug-in that encrypts the email, and sends an encrypted email package to a stock mail server.

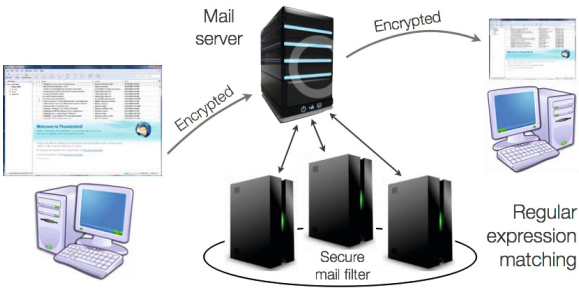

Fig. 3. Architecture of the Secure Mail Filter

We also created a "milter" plug-in for the mail server using the standard mail filter interface. The mail server automatically passes the encrypted email package to the plug-in, which is just a coordinator component that forwards the package to each of three cooperating share servers and awaits their responses.

As we shall see in a moment, the share servers each open the encrypted package (to the extent that they are able), extract random shares of the encrypted email, and together engage in a cooperative secure computation to analyze encrypted e-mail. When they have done their work, they return a random share of their boolean response to the plug-in, who XORs them together to obtain the mail filter response. If the answer is in the affirmative, the mail server forwards the message packet on to the recipient R. Otherwise, the mail server informs S of the rejection.

Communicating with the Share Servers. The sender S constructs an encrypted packet of data such that each of the recipients can extract exactly what they need, and no more than they should. In particular, neither the mail server nor the plug-in filter coordinating component should be allowed to know the content of the email. The three share servers A, B and C should each be able to obtain a random share of the original email, and the ultimate receiver of the email, R, should be able to read the whole thing—assuming the message is permitted through the email filter.

To accomplish all this, S uses a stream cipher encryption algorithm, Enc, such as AES in counter mode, together with a public-key system, Pub, such as RSA. S randomly generates three share-keys k_A, k_B and k_C, for the three share servers, and then computes a pseudo-random stream $\overline{k_A} = Enc_{k_A}(\overline{0})$ (the stream of zeroes encrypted using the stream cipher), and similarly computes pseudo-random streams $\overline{k_B}$ and $\overline{k_C}$. Using these streams as one-time pads, S creates a cipher text of the email message $CT = m \oplus \overline{k_A} \oplus \overline{k_B} \oplus \overline{k_C}$.

S now constructs and sends a package containing CT, together with targeted encryptions of the keys, namely $Pub_R(k_A, k_B, k_C)$, $Pub_A(k_A)$, $Pub_B(k_B)$, and $Pub_C(k_C)$, where $Pub_A(\_)$ is encryption using A's public key, and likewise for B, C, and R.

On receipt of the package, each of the servers A, B and C obtains the respective keys k_A, k_B and k_C (using their private keys), and now each can locally compute a copy of their designated pseudo-random stream: A computes $\overline{k_A}$ and B and C likewise. Using these streams, each of A, B, and C can construct a share of the original email message m: share $m_A = \overline{k_A} \oplus CT$, share $m_B = \overline{k_B} \oplus CT$, and share $m_C = \overline{k_C} \oplus CT$. The XOR ($\oplus$) of these three is the original message m as all the pseudo-random streams will cancel out.

Note than none of the servers are able to reconstruct m itself. In contrast, should the message pass the filter and be sent on, the recipient R will be able to reconstruct m, because it has been sent the keys that generate the three one-time pads.

The Secure Computation. The decision as to whether to send the email to the recipient or not is to be based on the result of evaluating a regular expression. For example, a filter for rejecting emails containing paragraphs with particular security markings might start to look something like this:

```
.*(((TOP|)SECRET)|TS|S)--SI--NO(CON|CONTRACTOR|FORN|FOREIGN).*
```

Each of the three share servers will know the regular expressions being used, but such details may be kept private from everyone else if desired.

There are many ways to evaluate regular expressions in plain text. For the secure setting we chose an algorithm based on non-deterministic finite automata (NFA), as opposed to selecting on the DFA algorithms. As every step of the algorithm has to operate over the whole of the state anyway (so as not to reveal which states are active), it makes sense to have many of those states active during computation[2].

For concreteness we used an efficient NFA algorithm that has been beautifully described in Haskell [FHW10]. The clarity of the description made it particularly easy to re-express the algorithm in our Haskell-based share language. We do not need to describe the algorithm in detail here. Suffice it to say that the algorithm uses a tree representation of the regular expression to represent the state, with each node of the tree flagged (or not) if the corresponding position in the regular expression is a match for the portion of the string consumed so far.

Fig. 4 shows an example for the regular expression `(xy)*x` after consuming just the input `"x"`.

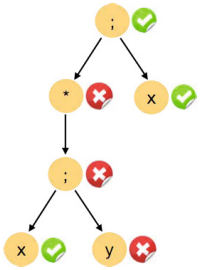

Fig. 4. Match-Annotated Regular Expression

For each new input character, the algorithm computes how to update the set of matched flags. That is, the matching function updates the flag-states on receipt of each new input character to produce a new flag-state for the computation on any remaining input characters. The flag corresponding to the top of the tree indicates whether the input so far has matched the entire regular expression.

3.2 Application 2: Secure VoIP Teleconference

For the second application we selected a client-server VoIP teleconference application that performs audio mixing of encrypted audio streams in real time.

[2] It would be interesting future work to explore the alternative choice: select a DFA algorithm, expand the NFA state set into a corresponding DFA state set (which can be significantly larger), and then use locality of the active state to gain amortized complexity improvements in the resulting secure computation.

As Fig. 5 shows, the architecture we used for the VoIP application is very similar to the mail filter application. This allowed us to re-use parts of the infrastructure even though the characteristics of the underlying computation were very different.

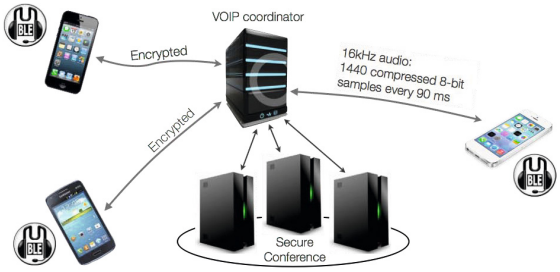

Fig. 5. Architecture of the Secure VoIP Teleconference

The client is a slightly modified open-source iOS-based implementation of the popular Mumble application[Mum], running on iPhone 5s, iPad Mini, and iPad Touch devices. The server is a modified open-source Linux-based implementation of the uMurmur VoIP server application, together with three share servers to perform the encrypted merges.

As with the mail filter setup, we communicate to the share servers by negotiating temporary keys, but with two differences. First, we negotiate temporary keys just once at the start of the audio stream and use the same keys throughout. Second, each client will generate a pair of keys for each server, one for the audio stream sent to the server, the other for the stream being received.

Each client samples audio into a 16kHz PCM data stream of 16-bit fixed point values. These are encoded by logarithmic compression to 8-bit uLAW samples. To tolerate processing and transmission latencies, the clients collect samples into 1440-sample packets, each packet containing 90ms of audio.

To transmit the audio, each client encrypts each audio packet by XORing the data with the XOR of the three pseudo-random streams, as with the mail filter. Similarly, the share servers each receive the data and extract their individual share of the audio packet by XORing it with their individual pseudo-random stream.

In each 90ms epoch, the share servers will compute multiple result streams—one for each client—by merging all the streams except for the client's own input audio stream. This saves us having to do echo-cancellation, but means the computation has to be repeated n times (for n clients). For each-8 bit packet of compressed audio, the computation is as shown in Fig. 6.

For each encrypted compressed sample in the packet, the share servers have to (1) decompress the sample to reform a 16 bit PCM sample, (2) add the decompressed value to the corresponding values in the packets from the other clients,

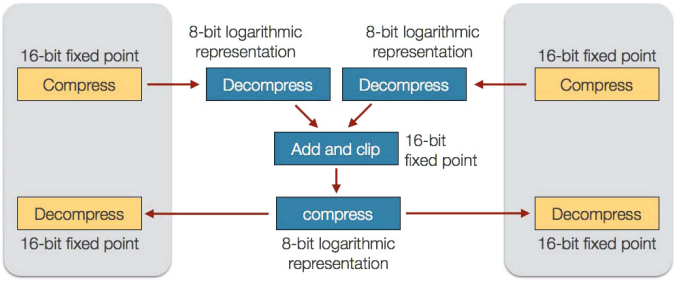

Fig. 6. Data Processing of Audio Packets

making sure that overflow or underflow are handled by graceful clipping, and (3) recompress the resulting 16-bit output audio sample into 8-bits for sending to the client. All of this has to be done cooperatively as the samples are encrypted throughout.

This process is repeated for each client packet received during the epoch. Thus for four clients, each share server has to perform 23,040 secure add-and-clip computations[3] every 90ms!

At the end of each 90ms epoch, the three share servers all XOR the result with the output key for each client, and send each result to the respective client. On receipt, each uMurmur client performs a matching decryption, and the samples in the resulting decrypted audio packet are uLAW decoded into 16 bit PCM format and inserted into a queue for audio playback.

4 Scaling the Secure Computation

Now that we have the structure of the applications, we turn our attention to ensuring the secure computation can scale to provide sufficient performance. Our notion of "sufficient" is not rigorous here; it is intended to reflect whether the results are even in the vicinity of being practical or not.

4.1 Secure Mail Filter

As with many EDSLs, the ShareMonad can produce many different kinds of interpretations of its "programs". One of the interpretations is an abstract representation of the arithmetic and/or logical "circuit" described in the ShareMonad program. In effect, it represents a partial evaluation of the program, leaving behind only the portion that needs to be executed securely.

As we noted earlier, in a step-by-step algorithm like regular expression matching—where each step consumes another input character—the circuit takes two kinds of input: the state of the computation from previous steps, and the new character being consumed. In turn it delivers a value representing the state

[3] 23,040 = 1440 samples × 4 input packets × 4 distinct audio result streams.

after this character has been considered. The updated state is used as the input state for the next character (Fig. 7). We also have shown extracting a boolean representing whether the whole regular expression has been matched.

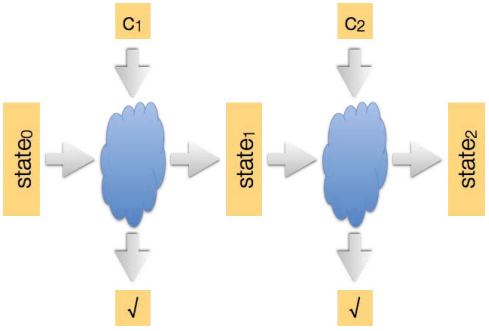

Fig. 7. Two Steps of the Recognizer

This is a raw circuit representing a single step of the recognizer. There is much we can do with the circuit to optimize it for execution. We group these in two phases: Simplification and Scheduling.

Simplification. The most expensive operation is AND (i.e. boolean "multiply"), so we apply many transformations to remove as many of these as possible. A representative set of simplifying transformations is shown in Table 1.

Table 1. Simplification Transformations

	Precondition	Before	After
Idempotence		$a \wedge a$	a
Factorization		$(a \wedge b) \oplus (a \wedge c)$	$a \wedge (b \oplus c)$
Constants	$c \neq d$	$(x = c) \wedge (x = d)$	F
Assoc. and commut.		$a \wedge (b \wedge a))$	$a \wedge b$
Redundancy	$a \Rightarrow b$	$a \wedge b$	a
Eliminate AND	$a \vee b$	$a \wedge b$	$\neg(a \oplus b)$

Most of the transformations are straightforward to implement. The last two deserve special mention, specifically because of the preconditions. These have to be proven to hold before the transformation is valid. We use the DepQBF solver [LB10] to verify whether the precondition holds, and only perform the transformation accordingly. Fig. 8 shows a small example of the kinds of improvements we get using these transformations.

In this case, the three ANDS we had before optimizations were reduced to one, the four state variables were also reduced down to one, and significantly, whereas

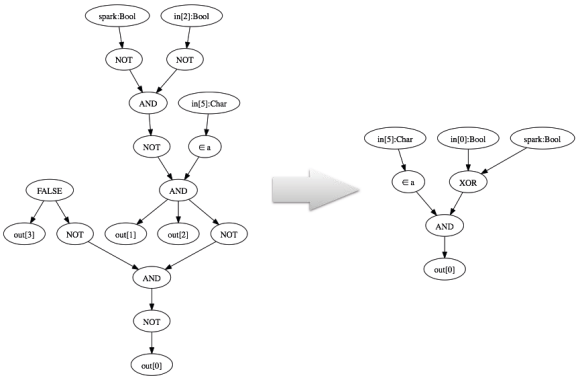

Fig. 8. Example of the Effects of Simplification

the original circuit would have required three rounds of communication, the optimized circuit only requires one. Obviously this is a very simplistic example, but the same kinds of result show up on much larger examples.

Unfortunately, the use of the logic solver is fairly time consuming (during compile time). To keep it manageable, we iterate it in the context of state-functions like the regular expression recognizer. That is, we optimize the circuit for one character; we then combine that circuit with itself to get a circuit for two characters (like in Fig. 7), which we then simplify and optimize. We then repeat the composition to get a circuit for four characters, then eight, and so on.

When do we stop going around this Simplify-Compose cycle? When we reach a point of diminishing returns. Fig 9 shows the effect of running this cycle over the recognizer circuit we get for a regular expression of the form:

```
.*(((TOP|)SECRET)|TS|S)--(ROCKYBEACH|STINGRAY).*
.*(((TOP|)SECRET)|TS|S)--SI--NO(CON|CONTRACTOR|FORN|FOREIGN).*
.*(((TOP|)SECRET)|TS|S|R|RESTRICTED)--(AE1|DS1|MT1|ST1)--LIMDIS.*
.*ac*cb.*
```

As the table shows, by the time we have composed two copies of the recognizer circuit the state is as small as it ever will be, but other measures are still improving. Through to the point where we have eight copies composed together, all the measures are still increasing by less than a factor of two, even though the input size is doubling. This starts to change in the transition from 8 to 16. At 16 copies of the recognizer, we have more than doubled the number of gates (because our heuristics are timing out on some of the larger circuits), and even the most crucial measure—the number of communication rounds—almost doubles too. Thus we can see that there is not much to choose between 8 or 16 copies of the recognizer, though we choose to use the 16 circuit because of the importance of minimizing the number of communication rounds. Multiple communication rounds causes the computation to stutter, introducing significant overheads.

	unoptimized				optimized			
input	ands	xors	state	comms	ands	xors	state	comms
1	203	0	358	10	149	15	119	4
2	388	0	358	12	277	27	117	5
4	756	0	358	14	493	53	117	6
8	1492	0	358	19	949	104	117	9
16	2964	0	358	33	**1,950**	212	**117**	**17**

Fig. 9. Optimization across Multiple Input Characters

Notably, our simplify-compose cycle has been very effective: have reduced the number of communications from 10 per character (unoptimized) to scarcely more than 1 per character.

Scheduling. It seems natural to perform each AND computation as early as its inputs become available. However, as Fig. 10 shows by graphing number of bits against communication round, this can lead to unbalanced communication patterns.

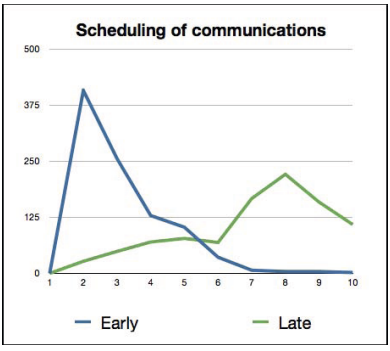

Fig. 10. Effect of Early vs. Late Scheduling

The graph shows an early spike in the number of bits being communicated (as many gates can be evaluated), with a long later tail in which very few bits are communicated. If we were just doing one computation this wouldn't matter as the number of bits is small, but we plan to do thousands of these together. In order to maximize flexibility in packing many copies of an execution together, we would like these communications to be as evenly balanced as possible. It turns out that the equally simple (but counter-intuitive) approach of scheduling each AND computation as late as possible produces less extreme peaks in the balance of communications, so we adopt this by default. It may be worth putting in additional effort to balance the communications more evenly still, but we have not done this.

Once we have scheduled the communications, we gather all the bits and pack them into 32 or 64 bit words in order to perform all the XOR and communication operations at the word level.

4.2 Secure VoIP Teleconference

When we turn our attention to the VoIP teleconference application, it turns out that the circuit characteristics are so different from the regular-expression circuits that we had to take a completely different tack.

Our first implementation was a direct implementation of the algorithm, where we decompressed the compressed audio samples to 16-bit values, added and clipped, and then recompressed. Unfortunately the result was running at about 12 seconds of computations for each 90ms audio sample!

The problem was in the combination of addition and clipping. Addition of 16-bit values can be done very efficiently so long as the values are stored as integers modulo 2^{16} (or larger). However, clipping required comparison operations. These are expensive unless the value is stored as a sequence of separate bits (i.e. not an arithmetic encoding). Whichever encoding is chosen, at least one of the operations is expensive.

We needed a different approach. We were able to take advantage of one significant characteristic of the computation: there are not many bits of input. The whole decompress-add-clip-recompress function on two streams takes 16 bits of input and delivers 8 bits of output. This is a classic opportunity for the oblivious lookup table we introduced previously [LAD12] (though we would have to work to make it scale well to 16 bits of input). The lookup table works as follows: we compute all possible values of the function in the clear, store them in the table, and perform shared access to the table at run time. The shared access works from randomized shares of the index value and delivers randomized shares of the table entry. In this case the whole secure computation reduces to oblivious table lookup.

Lookup Tables. Table lookup (i.e. simple array indexing) becomes tricky when no individual server actually knows what index to look up. Instead, each share server has a random share of the index value (i.e. a random value which if XORed with the random values from the other share servers would represent the real value). The servers have to do a cooperative computation to be able to obtain random shares of the the content of the table at the appropriate location.

Note that the lookup algorithm has to act on all the entries of the table otherwise a server 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 have a cooperative `demux` protocol that maps a binary representation of a value into a linear, unary representation.

In plaintext, a 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. So, for example, with the convention that the demux bits are numbered from left to right:

```
demux 0000 = 1000000000000000
demux 1000 = 0000000010000000
demux 1111 = 0000000000000001
```

and so on.

Still in the plaintext version, the table lookup is now just a kind of "inner product" between the result of the demux function and the table itself (see Figure 11), where the multiply operation is bit-masking. The result of the demux is used to mask the corresponding table entry (i.e. return the entry or 0), and the results across the whole table are XORed together. 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.

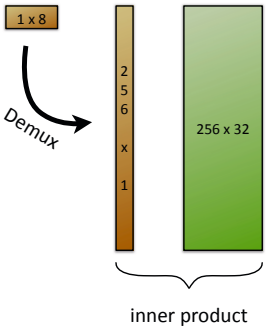

Fig. 11. Inner Product with Demux

We now simulate the plaintext algorithm with a randomized share version. The shared demux computation would map a *share* of a 4-bit value, to a *share* of a 16-bit value. That is, if $x = x_A \oplus x_B \oplus x_C$, if $d = demux(x)$, and if d_A, d_B, and d_C are the result of running the demux protocol on the x_i's, then $d = d_A \oplus d_B \oplus d_C$. For example, if we compute the demux of `0x8`, again going from 4-bits to 16-bits, then (subject to randomness) the d_i might be as follows:

```
d_A = 1011001011101011
d_B = 0011010011001101
d_C = 1000011010100110
               ^
```

Notice that only the indicated 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. Each d_i is a randomized share of the true demux d. That is, for each bit position j in the demux shares, $d_A(j) \oplus$

$d_B(j) \oplus d_C(j) = d(j)$. Thus all these XORs will be 0 except for the single bit position corresponding to the original index, which will have value 1. The mask operation of the "inner product" function (written here as M) distributes across $\oplus$, so that $M(d_A(j) \oplus d_B(j) \oplus d_C(j), e) = M(d_A(j), e) \oplus M(d_B(j), e) \oplus M(d_C(j), e)$. This means that we can compute the inner product operations locally on each share machine. Demux is the only part that needs to be computed cooperatively.

Demux. In plaintext, demux can be expressed as a divide and conquer algorithm, satisfying the equation $demux(bs{+}{+}cs) = demux(bs)\#demux(cs)$, where $+{+}$ is sequence concatenation, and # is cartesian product on sequences of bits.

For example, if `demux "10"` is given by `"0010"` and `demux "01"` is given by `"0100"`, then `demux "1001"` is given by `"0000 0000 1000 0000"`, which is the linearization of the product table between the two.

In our previous work, we had expressed the cartesian product as a parallel multiply by expanding each of the smaller demuxes into structures the same size as the result [LAD12]. The advantage was that we could just use the generic multiply protocol. The downside was that the amount of communication is proportional to the size of the final demux. This was fine for small tables (we were previously only doing lookup tables with 256 elements), but now our tables are starting to become large (with 65536 elements), and the communication bandwidth dominates.

We note that bit-level cartesian product (#) distributes over XOR ($\oplus$) just like AND (&) does, so the multiplication table is identical to the table for AND. We replicate the implementation of shared multiply—but using cartesian product on bits sequences—to produce a direct implementation of shared cartesian product. This means that our communications grow much slower than before. In fact, for a table with 2^n entries, we require $log(n)$ communications, communicating $O(2^{n/2})$ bits. In this case, where n is 16, we have 4 rounds of communication, and around 700 bits per server being communicated.

The cartesian product # operation is specified recursively using the divide and conquer pattern above. We find it valuable to leave the final expression unexpanded. That is, if p_i and q_i are the randomized shares of the demuxes of the upper and lower 8 bits of the original 16-bit index, then the shares of the outermost call of # returns the value $(p_A \# q_B) \oplus (p_B \# q_B) \oplus (p_B \# q_A)$, and correspondingly for the other shares. Instead of computing the final # we create an abstract representation of the computation, or rather of $(p_A \# q_B) \oplus (p_B \# (q_B \oplus q_A))$. We can use this unexpanded definition of # to act as a pair of 2-dimensional indices into the table, as indicated in Fig. 12. This unexpanded definition of # reduces the size of the demux value used in the "inner product": we now demux two 256-bit values directly, instead of constructing one large 65535-bit value.

In exchange for the not having to construct the 64k value explicitly, we must perform twice as many local XORs as we have to compute the "inner product" of the table twice. As before, we use the indices to mask out table entries and XOR the remainder. This calculation over the table requires 2×64k operations, which can still be expensive even though they are purely local. We have a further

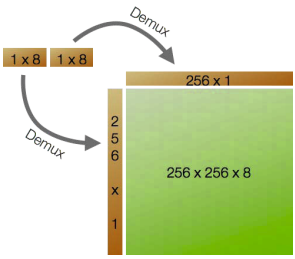

Fig. 12. Two Dimensional Demux

optimization whereby we chunk the table in four rows at a time, and precompute the selective XORs of these rows. This expands the table by a factor of 4, but cuts the table computation time in half.

5 Assessment

Our goal was to test whether we were able to scale secure computation to the levels required by applications. This is a fuzzy standard, but we can still do qualitative assessments against it. We assess architecture, algorithmics, and performance.

The architecture and infrastructure aspects of secure computation were able to be integrated well. In both applications, despite having many different timing and structure characteristics, we were able to adapt the application server to interact with a secure computation engine in order to perform the core operations securely. The bandwidth and latency requirements between the client(s) and the server were scarcely altered.

Regarding algorithmics, the mail filter application was surprisingly easy. We had to apply careful thought to find a version of the algorithm that would suit the oblivious computation world, but once selected, the conversion to use secure flags rather than plaintext flags was straightforward. This would not have been the case if the algorithm used the flags to determine where to branch, but for us it did not.

The VoIP application was tougher. Our first transcription of the algorithm into the secure computation world was so slow that we initially despaired of ever getting it to be relevant. However, the fact that it operates on small data items turned out to be crucial. Once we thought to express the core of the algorithm as a table lookup, the expression of the algorithm became trivial, though we still had to work hard to get performance.

As for performance, we have to conclude that we are only just reaching the point of usability. In the mail filter case, we are able to send a 1 page email, analyze it with the regular expression described earlier, and obtain a response in 30-60 seconds. We believe that there are a number of improvements we could still apply (including increasing the use of parallel processing) that could reduce

this by up to another order of magnitude perhaps, at which point it is indeed starting to become practically relevant.

For the VoIP teleconference application, we conducted experiments both in Oregon and Virginia, hosting our servers in the Amazon EC2 cloud instance geographically closest to each experiment. In the first experiment, we conducted audio teleconferences with up to four clients, using spoken voice as the audio content. Audio was reliably understandable by all participating speakers, though we noted the presence of audible clicks and other artifacts. In the second experiment, we streamed recorded music into an iPad Mini client via the device microphone, and an audience of approximately 60 listened to the output audio stream on a second client, an iPhone 5s. Except for occasional distortion corresponding to spikes in network latency, audience members noted that audio quality was good, approximating what might be expected of broadcast radio.

6 Related Work

The classic "real world" example of secure computation is a Danish beet auction in 2008 [BCD+08]. There, 1200 Danish farmers submitted randomized 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 confidentially built into the MPC protocols provided sufficient reassurance to the famers, 78% of whom agreed that, "it is important that my bids are kept confidential."

Our table lookup has many aspects in common with private information retrieval (PIR) algorithms [CGKS95], except that we are working with peer machines rather than a client querying a distributed database. The $O(\sqrt{n})$ growth in communication bandwidth we see (where n here is the size of the table, not of the index), is directly comparable to that of PIRs. It will be interesting to see whether the peer case can be conveniently generalized to more servers as with PIRs.

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 ShareMonad 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.

The SPDZ system [DKL+13] uses a similar computation model, except that it works with precomputed multiplication triples. This provides two advantages: it allows the online computation phase to work with any number of parties, and it provides for covert security (a cheating party is extremely likely to be caught).

The relative performances of Sharemind, SPDZ and our ShareMonad are hard to determine with accuracy, but there is some evidence they are all within factor of two of each other, which in this world means roughly comparable (given that we are all still discovering order of magnitude improvements!).

7 Conclusion

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 network communications and coordination required. In Yao garbling, the expense arises because conjunctions are represented by encrypted gate tables that have to be created, 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 noise reset operations, which entail evaluating a homomorphic encrypted instance of the decrypt operation.

When optimizing computations in MPC 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, all the MPC 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 continues to explore the kind of algorithmic rethinking and compiler transformation that are required, but much more is needed before secure computation is fully practical.

References

[BLW08] Bogdanov, D., Laur, S., Willemson, J.: Sharemind: a framework for fast privacy-preserving computations. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 192–206. Springer, Heidelberg (2008)

[BCD+08] Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M., Toft, T.: Secure Multiparty Computation Goes Live. In: Dingledine, R., Golle, P. (eds.) FC 2009. LNCS, vol. 5628, pp. 325–343. Springer, Heidelberg (2009)

[CGKS95] Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private Information Retrieval. In: Proc. of IEEE Conference on the Foundations of Computer Science (FOCS) (1995)

[DKL+13] Damgaard, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.: Practical Covertly Secure MPC for Dishonest Majority or: Breaking the SPDZ Limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 1–18. Springer, Heidelberg (2013)

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

[G04] Goldreich, O.: Foundations of Cryptography. Basic Applications, vol. 2. Cambridge University Press (2004)

[LAD12] Launchbury, J., Adams-Moran, A., Diatchki, I.: Efficient Lookup-Table Protocol in Secure Multiparty Computation. In: Proc. International Conference on Functional Programming (ICFP) (2012)

[LB10] Lonsing, F., Biere, A.: DepQBF: A Dependency-Aware QBF Solver. JSAT 7, 2–3 (2010)

[Mum] `http://mumble.sourceforge.net`

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

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

[FHW10] Fischer, S., Huch, F., Wilke, T.: A Play on Regular Expressions: Functional Pearl. In: Proceedings of the International Conference on Functional Programming, ICFP 2010 (2010)