An Architecture for Practical Actively Secure MPC with Dishonest Majority

Marcel Keller
Department of Computer Science, University of Bristol
m.keller@bristol.ac.uk

Peter Scholl
Department of Computer Science, University of Bristol
Peter.Scholl@bristol.ac.uk

Nigel P. Smart
Department of Computer Science, University of Bristol
nigel@cs.bris.ac.uk

ABSTRACT

We present a runtime environment for executing secure programs via a multi-party computation protocol in the preprocessing model. The runtime environment is general and allows arbitrary reactive computations to be performed. A particularly novel aspect is that it automatically determines the minimum number of rounds needed for a computation, given a specific instruction sequence, and it then uses this to minimize the overall cost of the computation. Various experiments are reported on, on various non-trivial functionalities. We show how, by utilizing the ability of modern processors to execute multiple threads at a time, one can obtain various tradeoffs between latency and throughput.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*security and protection*; D.4.6 [**Operating Systems**]: Security and Protection—*cryptographic controls*

Keywords

Multi-Party Computation

1. INTRODUCTION

Recent years have seen great advances in practical variants of protocols for secure Multi Party Computation (MPC). Initially these practical instantiations came in the restricted security model of dealing with passive (semi-honest) adversaries [3, 6, 25, 32, 9]. However, more recent work has focused on the case of full active security [5, 11, 15, 26, 27], or sometimes covert security [13].

MPC protocols come in two flavours: one flavour based on Yao circuits and one flavour based on secret sharing. In those based on Yao circuits (which are predominantly focused on the two party case) a function to be evaluated is expressed first as a boolean circuit. The boolean circuit is then "encrypted" by one party in a process known as garbling. The encryptor then passes the encrypted circuit over to the other player, who then evaluates the circuit on their own input. The input of the evaluating player is obtained from the encryptor via an oblivious transfer protocol.

The first implementation of the Yao approach in the semi-honest setting was the FairPlay system [25], which was followed by an implementation in the active security case [24]. Following the work in [28] it has been common to benchmark such Yao based implementations against evaluating the AES circuit, see for example [16, 17, 23, 31]. The state-of-the-art in terms of Yao based computation would currently appear to be the system in [20].

In the secret sharing based approach initial practical work focused mainly on the case of three parties with only one allowed semi-honest adversary [6, 32]. Systems such as [11, 32] could cope with larger numbers of players, and more powerful adversaries, but only with a significant degradation in performance. Recent practical MPC work in the secret sharing tradition has focused on full active security against a dishonest majority, utilizing a form of "shared MAC" to provide security [5, 13, 15, 26]. It is in this area that our work is focused. Whilst secret sharing based approaches are not as well tailored to evaluate functions such as AES, the AES functionality is still used as an important benchmark in this area [12, 13, 21, 22].

Another aspect of modern practical work on secret sharing based MPC protocols is the fact that they are often presented in a "preprocessing model". In this model one is allowed to perform in an "offline phase" the generation of random data, which will be consumed in the "online phase". The key aspect is that the data produced in the offline phase must be generic and not depend on the inputs to the function (which are unknown until the online phase), nor too tied to the specific function being evaluated (which again may not be known until the online phase commences).

The final consideration is whether the protocol supports fully reactive computation. By this we mean that, after a function result has been determined, one can execute another function; with the precise choice of the new function depending on the output of the first computation. A fully reactive computation allows this process to be continued ad infinitum, including possibly executing the offline phase again if the preprocessed data runs out.

Previous implementation work, in the secret sharing setting, has mainly focused on the underlying MPC protocol and how to perform specific pre-defined functionalities; for example, the AES circuit above. The traditional approach is to express the function to be evaluated is described as an arithmetic circuit and then this circuit is evaluated by executing a sequence of addition and multiplication "gates". This is a throw back to the origins of MPC in the theoretical community; where functions are always represented as arithmetic circuits.

In this work we focus on modern secret sharing based MPC protocols in the pre-processing model whose online phase is actively secure, for example the SPDZ protocol [13, 14, 15] or the Tiny-OT protocol [26], or even the earlier information theoretic based pro-

CCS 2013 Berlin, Germany
ACM 978-1-4503-2477-9/13/11.
http://dx.doi.org/10.1145/2508859.2516744.

tocols such as VIFF [11]. In practice such protocols *do not* execute arithmetic circuits; they can evaluate far more elaborate basic operations since they have the ability to selectively "open" secret data. In fact if the pre-processed data is so called "Beaver Multiplication Triples", then a multiplication gate is actually implemented as a sequence of local additions and selective openings; and of course consuming some pre-processed data.

Shifting ones viewpoint to treating the three operations of local additions, "open" and consuming pre-processed data as the basic operations enables far more expressive operations. This viewpoint is indeed implicit in the work of [10] which examines protocols for comparison, bit-extraction and exponentiation amongst others. The key ability utilized, in its most basic form, is to enable a secret value to be masked, and then the masked value to be opened. Whilst this might seem a small additional functionality, it turns out in practice that such an operation allows one to perform highly complex tasks without needing to resort to their description as an arithmetic circuit.

Shifting ones view point in this way is also important from a practical perspective as the main bottleneck in implementations is the scheduling of the open instructions. By focusing on "open", as opposed to multiplication, as the key operation to optimize one can achieve significant performance improvements.

In this work we present the design of an efficient runtime execution environment for the protocols in the SPDZ family [13, 14, 15], much of our work will extend to other protocols such as TinyOT [26]. This protocol family supports fully malicious (active) or covert security, is in the preprocessing model and can (using the techniques of [14]) support fully reactive computation. Unlike prior work we concentrate only on the online phase and show how to design efficient compilers and a runtime environment to enable high level operations. Prior implementation work on the SPDZ family has either concentrated on the specific functionalities over small finite fields, such as evaluating AES [13], or has focused on the more mathematically complex offline phase [14, 15].

Our focus is on more general purpose computation, and in particular how to enable a general purpose high level function description, which is then translated into low level operations, which are then executed by an MPC runtime engine. In particular we present experimental results for an actively secure online phase for various high level functionalities such as integer multiplication and comparison, fixed point and floating point operations as well as runtimes for AES. Our AES runtimes are significantly better than prior runtimes for AES using other actively secure MPC implementations.

Our prime motivation is to understand how to engineer the overall runtime environment, rather than the design of specific protocols. We therefore utilize prior work of others for our higher level functions, i.e. for integer comparison we use [7], for fixed point numbers we use [8], for floating point numbers we use [1], for AES we use the techniques of [13]. Whilst much of our design is biased towards the SPDZ protocol, many aspects can be directly extended to other MPC protocols that only involve broadcasting shares to all parties for communication. This is true for all protocols based on Beaver's technique [2]. However, the basic idea of automatically optimizing the communication pattern is also applicable to protocols based on Shamir secret sharing such as [4]. Other work, e.g. [19, 29], on optimization for MPC compilers has focused on issues related to what information is leaked when values are revealed to enable branching; this is an orthogonal question to those that we consider.

We present our experimental results in the following way. We show how one can trade, for the different high level functionalities, latency for throughput. In many previous works on MPC the focus has been on throughput or latency alone. This however is not realistic; in a given application scenario one is likely to have a specific latency and a specific throughput one has to meet for the MPC solution to be applicable. We show how, by batching different numbers of operations together, and by running in different numbers of threads, one is able to trade throughput for latency. This allows one to reach a specific *sweet spot* for a given application.

In summary, existing MPC implementations do not provide active security against a dishonest majority, do not scale (efficiently) to any number of parties, require knowledge of (and manual optimizations for) the underlying MPC protocol, or are generally not very efficient. To our knowledge, we present the first MPC implementation that achieves the highest level of security in the online phase, efficiently scales to at least five parties, provides a high-level language that does not require a detailed understanding of MPC, and is competitive with previous implementations achieving the same level of security. To accomplish this, we use the SPDZ protocol for the desired security and scalability, and we have implemented a compiler for a Python-like language that optimizes the communication pattern of the program. The compiler outputs a bytecode that can be interpreted by a virtual machine. For efficiency, this virtual machine is implemented in C++ and kept relatively simple. We believe that our approach optimally combines usability with the efficiency gains of using machine code and avoiding idle time caused by the multi-round nature of MPC based on arithmetic circuits.

2. OVERVIEW OF THE SPDZ PROTOCOL

At the core of our work is a multiparty computation protocol in the preprocessing model, with specific properties. In particular we assume the preprocessing produces so-called Beaver Multiplication Triples. These are triples of secret shared random values, such that the third random value is the product of the first two. In addition we assume the ability to produce other forms of pre-processed data. For more details see below.

There are a number of modern protocols in this family, and we will focus on those in the SPDZ ("Speedz") family, [14, 15]. This family is an extension of a protocol first introduced by by Bendlin *et al.* [5]. The advantage of this approach is that expensive, public-key machinery can be offloaded into the preprocessing phase. The online phase uses (essentially) only cheap, information-theoretically secure primitives and can therefore be extremely efficient. Moreover, the technique comes with extremely strong security guarantees: the online phase is actively secure against a dishonest majority of corrupted players, irrespective of the security of the offline phase. Thus we obtain a protocol which is actively (resp. covertly) secure if we combine our online phase with an offline phase which is actively (resp. covertly) secure as in [15] (resp. [14]). In addition our online phase can evaluate reactive functionalities (i.e. multiple sequential programs) and not just execute secure function evaluation.

Throughout this exposition we assume a number of players n and a finite field $\mathbb{F}_q$ over which computations will be performed have been fixed. The basic data type used is a secret sharing scheme over the field $\mathbb{F}_q$. Each party in the protocol P_i has a uniform share $\alpha_i \in \mathbb{F}_q$ of a secret value $\alpha = \alpha_1 + \cdots + \alpha_n$, thought of as a fixed MAC key. We say that a data item $a \in \mathbb{F}_p$ is $\langle\cdot\rangle$-shared if P_i holds a tuple $(a_i, \gamma(a)_i)$, where a_i is an additive secret sharing of a, i.e. $a = a_1 + \cdots + a_n$, and $\gamma(a)_i$ is an additive secret sharing of $\gamma(a) := \alpha \cdot a$, i.e.

$$\gamma(a) = \gamma(a)_1 + \cdots + \gamma(a)_n.$$

2.1 Offline Phase

In this section we summarize the data produced during the offline phase of the SPDZ protocol that is required by our runtime. The main purpose of the preprocessing phase is to generate multiplication triples, namely shares of random values $\langle a \rangle, \langle b \rangle, \langle c \rangle$ satisfying $a \cdot b = c$, which are used to multiply secret shared values during the online phase using a technique of Beaver [2].

Whilst multiplication triples are the only data theoretically needed for the online phase, we can also use the offline phase from [14, 15] to create additional raw data that improves the efficiency of certain operations. In Table 1 we detail all data that is used during the online phase in our various higher level functions. The input data is used to enable parties to input values into the computation, see [15], we shall not discuss this further in this paper. The square data is used to perform squaring of shared values more efficiently than general multiplications; this is useful in general computation but does not feature in our specific examples later in the paper. The inv tuples are used for constant round protocols for integer comparison; we shall see later that the logarithmic round protocols are actually more efficient in practice for the data types we are interested in.

Command	Outputs	Properties
triple	$\langle a \rangle, \langle b \rangle, \langle c \rangle$	$a, b \xleftarrow{\$} F_p, c = a \cdot b$
square	$\langle a \rangle, \langle b \rangle$	$a \xleftarrow{\$} F_p, b = a^2$
bit	$\langle b \rangle$	$b \xleftarrow{\$} \{0, 1\}$
input	$\langle r \rangle$	$r \xleftarrow{\$} F_p$
inv	$\langle a \rangle, \langle a' \rangle$	$a \xleftarrow{\$} F_p, a' = a^{-1}$

Table 1: Data Prepared During the Offline Phase

The offline phase is assumed to produce five files of pre-processed data for each player. Each file containing the relevant shares of the above data. Exactly how much of each type of data one needs to compute in the offline phase is of course unknown until the exact function one is computing is known. However, one can make a heuristic over estimate of this number. In addition as soon as data produced in the offline phase has been exhausted one can return to the offline phase to compute some more (note this option is only available if one is using the MAC checking strategy described in [14]).

2.2 Online phase

During the online phase the parties compute on open values (i.e. values which are not secret shared) and secret values (i.e. values which are secret shared). All standard arithmetic (+, ×, /, %) and bitwise boolean ($\vee$, $\wedge$, $\oplus$) operations can be performed on open values; where for the latter boolean operations we assume a fixed binary representation of values within the finite field $\mathbb{F}_q$. The arithmetic operations can also be applied to one value which is shared and one value which is opened, resulting in a new value which is shared. Finally, in terms of arithmetic operations, two shared values can be added together to result in a shared value. Another set of operations deal with loading data from the files precomputed in the offline phase. These are all shared random values, subject to the constraints given in Table 1.

All of the prior operations are "local", i.e. they do not require any interaction between the parties. As these local operations which produce shared values are performed, the parties can locally update the value of their shares of the associated MAC value. The power of the online phase comes from the ability to interactively "open" shared values so as to produce opened values. In such an operation a shared value $\langle a \rangle$ is opened to reveal a to all parties, but the associated shared MAC values, $a \cdot \alpha$, are kept secret.

To protect the parties against malicious parties the shared MACs of all such opened values need to be checked for correctness, and this must be done in a way so that neither the MAC key nor the MAC value itself is revealed. This is done using the protocol described in [14]. We execute this protocol so that batches of MAC values are checked all at once; in particular we batch up to 100,000 such MAC values to be checked into one operation. Thus the MAC checking protocol is executed after every 100,000 such openings, and once at the end of the computation to check any remaining values.

In [15] the following $O(n)$ protocol is suggested for performing the opening operation on the shared value $\langle a \rangle$.

- The players pick an arbitrary "nominated player", say P_1.
- Players P_i for $i = 2, \cdots, n$ send P_1 the value a_i.
- P_1 computes $a = a_1 + \cdots + a_n$ and sends a to all other players.

It is readily seen that the communication complexity of the above protocol is that each player (on average) sends a total of three field elements per opening. However, in practice we have found the following naive protocol to be more efficient.

- P_i sends a_i to all other players.
- Each player locally computes $a = a_1 + \cdots + a_n$.

Here the parties need to each send $n-1$ field elements per opening, which will be worse than the prior method as soon as $n > 3$. However, in practice we found that the naive method performs better than the $O(n)$ method for $n \leq 5$. This is because it is a one round protocol as opposed to a two round protocol. This dependence of efficiency on rounds will be a feature of our work and we will take a great deal of trouble in later sections in minimizing the round complexity of the entire computation in an automatic manner. Clearly to minimize rounds across a computation many different values to be opened can be opened at once in either of the methods above.

From the above description of the basic functionalities of the online phase it is not clear whether such a runtime is universal, i.e. can compute any arbitrary function. Since every function can be expressed as an arithmetic circuit over $\mathbb{F}_q$ and addition of shared values is a local operation, all that remains to show universality is to demonstrate that multiplication can be performed. Given two shared values $\langle x \rangle, \langle y \rangle$, to multiply them we take a precomputed multiplication triple $\{\langle a \rangle, \langle b \rangle, \langle c \rangle\}$ and then the parties open the values $\alpha = x + a$ and $\beta = y + b$. The shared value $\langle x \cdot y \rangle$ can then be computed from the local operation

$$\langle x \cdot y \rangle = (\alpha - \langle a \rangle) \cdot (\beta - \langle b \rangle) = \alpha \cdot \beta - \beta \cdot \langle a \rangle - \alpha \cdot \langle b \rangle + \langle c \rangle .$$

Thus multiplication is itself composed of our three primitive operations of local additions, open and consuming pre-processed data.

3. VIRTUAL MACHINE

To implement the online protocol we took a 'ground up' approach to architecture, starting with a very simple register-based virtual machine that can be run on an ordinary desktop computer by each party. By focussing initially on the low level design details we have obtained a high performance implementation with minimal overhead, leaving decisions for optimization and compilation to a higher level. Our virtual machine could be used to run code from a variety of language descriptions, given a cross-compiler to output the appropriate bytecode. To demonstrate this we use a simple language based on Python for all our programs. Details of the compilation process and special-purpose optimizations that we developed to reduce communication costs are given in Section 4.

Programs to be run by the virtual machine must be encoded into bytecode as assembly-like instructions (generally produced by a higher level compiler), which are then parsed and executed. To make the most of modern multi-core processors, we allow a number of such precomputed bytecode files to run simultaneously, each in a separate thread of the runtime virtual machine. This then leads to an additional level of complication in relation to how the bytecode files consume data from the offline phase, how the bytecode files are scheduled across a large computation and communicate with each other, and how branching and looping are to be performed. In this section we describe the low level design decisions which we have made to solve these problems. To ease exposition we shall refer to a single sequence of bytecode instructions as a *tape* in the following presentation.

The virtual machine consists of a memory holding a fixed number of clear and secret shared values. This memory is long term, and as the virtual machine starts up/shuts down the memory is read from/written to the player's local disk. In addition the virtual machine runs a series of $t + 1$ threads; one thread acts as a control thread while the other t threads have the task of executing (or not) a tape of bytecode. The t tape-running threads open pairwise point-to-point communication channels with the associated threads in the other players' runtime environments. There is no limit to the number of concurrent tapes, t, but in practice one will be restricted by the number of cores. For our test machines with eight cores, we found best performance when t is limited to seven or fewer.

The control thread executes the main program, or *schedule*. This is a program which specifies exactly which tape should be run at which point. The schedule also specifies which, and how many, tapes are executed in parallel; with up to t such tapes may be executed in parallel. The control thread takes the tapes to be executed at a given step in the program, passes them to the execution threads, and then waits for the threads to finish their execution, at which point the control thread processes the next stage of the program. Communication between tapes is done via reading and writing to the virtual machine's long term memory. To avoid unnecessary stalls there is no locking mechanism provided to this memory. So if two simultaneously running threads execute a read and a write, or two writes, to the same memory location then the result is undefined since it is unspecified as to which order the associated instructions will be performed in. Thus in this respect we have traded speed for safety at the lowest level; it is the task of the compiler or programmer to ensure the safe behaviour of concurrently running bytecode.

We now turn to discussing the execution threads and the associated bytecode instructions within each tape. Each execution thread has its own local shared and clear memory. To avoid confusion with the long term memory we shall call this local memory a *register file*, referring to the values as shared or clear *registers*. The values of a register are not assumed to be maintained between an execution thread running one tape and the next tape, so all passing of values between two sequential tape executions must be done by reading and writing to the main virtual machine memory. Again, whilst this may seem like a very primitive approach to take, we are ensuring that there is no unnecessary overhead and bloat in the virtual machine, leaving more complex elements to the next level up.

The bytecode instructions within a tape are influenced by the RISC design strategy, coming in only three basic types; corresponding to the operations described in Section 2.2. The first of these most closely resembles traditional assembly code instructions, consisting of load and store operations for moving registers to and from the long term memory, and instructions for executing *local* arithmetic and bitwise logical operations described in Section 2.2, that can be performed without interaction. All of this class of bytecode instructions are of the simple one output, one or two input variety found in modern processors, where the inputs may be registers or immediate values and the output register type depends on the input types as specified earlier. The other two types of instructions are data access instructions, which serve to load data from the preprocessing phase, and a special instruction to open shared register values into clear registers. Note that this last instruction is the only one requiring interaction between players. We now turn to discussing each of these special MPC instruction types in turn.

The instructions for loading data from the preprocessing phase are denoted `triple`, `square`, `bit`, `input` and `inv`, and they take as argument three, two, one, one and two shared registers respectively. The associated data is loaded from the precomputed data and loaded into the registers given as arguments. This leads to the problem that multiple concurrent tapes could read the same data from the precomputed global store. Thus the control thread on passing a tape to an execution thread for execution first calculates a (reasonable) upper bound on the amount of pre-processing data it will consume before it is executed. This can be easily done in the case of programs with no loops by simply counting the number of such instructions in the tape. Then the control thread allocates a specific portion of the unused precomputed data to the thread for use within this tape. This solution avoids complexity in the offline phase (by not requiring it to produce separate data for each thread) or added complexity to the online phase (by not requiring a locking mechanism to access the precomputed data), both of which would negatively affect program complexity and/or performance. The cost for this simple solution comes when executing tapes containing branching instructions, an issue we return to in Appendix A.

The process of opening secret values is covered by two instructions. The `startopen` instruction takes as input a set of m shared registers, and `stopopen` an associated set of m clear registers, where m can be an arbitrary integer. They then execute the protocol from Section 2.2 to open the shared registers and place the results into the designated clear registers. The fact that we can perform m of these operations in parallel is vital in improving the throughput and latency of applications run on the virtual machine. Furthermore, splitting the process in two steps allows one to execute local instructions while the runtime is waiting for data from other parties to arrive.

4. THE COMPILATION PROCESS

To produce the bytecode tapes required by our virtual machine, we developed a compiler for a simple high-level language derived from Python. The compiler takes a Python-like program description, extracts individual threads from this and breaks each thread down into basic blocks. The individual basic blocks are then optimized and each thread used to produce a bytecode tape for the virtual machine. Our optimization guarantees that, within a basic block, the program will be compiled with the *minimum possible* number of rounds of communication. This, coupled with the familiarity to many of the Python programming language, ensures that the workload of implementing complex MPC functionalities is eased considerably.

Instead of the traditional approach of using parsing and lexing tools for compilation, we chose to implement the core language features as a Python library, so that programs can be executed by

Python itself, which then produces the relevant bytecode. This allows for (compile-time) access to many of Python's powerful features such as list comprehensions, classes and functions with relatively little effort. The library consists of a set of functions for creating, storing and loading clear and secret shared data types; loading preprocessed data and more complex features such as threading and branching. We then use operator overloading to provide transparent support for arithmetic functions on our data types.

Upon execution, the Python code creates bytecode instructions from our library functions and the overloaded arithmetic operators, whilst keeping track of the appropriate threading and basic block information. Complex functions such as secret multiplication and comparison are expanded into sequences of the basic RISC-like instructions described in Section 3. Note that at this stage the open instructions will only have one shared and one clear register as arguments.

So despite our RISC instructions not supporting complex operations such as the multiplication of two shared values in $\mathbb{F}_q$, we can easily implement this using the compiler. This abstraction is key to our approach; we do not at the low level execute a program corresponding to an arithmetic circuit over $\mathbb{F}_q$ as in other approaches. We instead execute only a sequence of operations which are linear operations on the shared values, open instructions, or load precomputed data. All high level operations can be built up from these instructions; and indeed more efficient high level operations can be built which do not involve reduction to arithmetic circuits.

Using the above ideas we arrive at a point where we have produced a set of bytecode tapes in static single assignment (SSA) form, with an unbounded number of clear and shared registers, and standard compiler optimizations such as common subexpression elimination can be performed if required. We then optimize the communication complexity of each tape by reordering and merging the open instructions, as follows:

- Calculate the *program dependency graph* $G = (V, E)$, whose vertices correspond to instructions in the bytecode, and directed edges represent dependencies between instructions.

- Let $T \subseteq V$ denote the set of startopen vertices and let S be the subset of T with no preceding nodes in T.

- Since all nodes in S are independent of each other, we now merge them together into a single source node (and instruction) s, and do the same with associated stopopens.

- To compute the remaining merges that can be done, assign weights to edges as follows:

  $$w_{i,j} = \begin{cases} -1 & \text{if } v_j \in T \setminus S \\ 0 & \text{otherwise} \end{cases}$$

  This ensures that the length of a path between any two nodes is exactly -1 times the number of startopen nodes lying on the path, excluding the first node and any source startopen nodes.

- For each vertex v, compute dist_v, the shortest distance from the merged source s to v. Since G is a DAG, this can be done in linear time with a standard shortest paths algorithm using dynamic programming.

- Merge all startopens (and their associated stopopens) that are the same distance from s (by the shortest path property, these must all be independent).

After this step, all independent startopen and stopopen instructions in the original program have been merged together. We can now compute a topological ordering of G to produce a program with the minimal number of rounds of communication for the given instruction sequence.

To illustrate the benefit of this, we implemented a standard polynomial approximation for the floating point $\sin(x)$ function. If this is compiled without merging open instructions (bar those merges which are automatically encoded in the lower level constituent operations) one would obtain a round complexity of 5874. On application of our optimizer the number of rounds reduces to 759.

When computing the topological ordering, we calculate which round of openings each instruction depends on and which round of openings depends on the instruction. This can be done similarly to the shortest paths method above, traversing G in reverse order. Instructions that do not have a dependency in both directions, like memory operations, are placed close to the remaining dependency or close to the beginning if there is no dependency at all. This reduces the memory usage of the virtual machine because it reduces the lifetime of registers.

Finally, there are instructions that have dependencies in both ways, i.e. those that depend on some opening round i, while at the same time opening round $i + 1$ depends on them, clearly have to be placed between the two rounds. However, instructions that only depend on rounds up to i and are independent of round $i + 1$ can be placed between the startopen and stopopen of round $i + 1$. This utilizes the time used for waiting for data from other parties for performing local computations.

To complete compilation, the compiler allocates registers. Since all programs are in SSA form at this point, the allocation is straightforward. The compiler processes the program backwards, allocates a register whenever a register is read for the last time, and deallocates it whenever a register is written to. This ensures that a minimal numbers of registers are used, which minimizes memory usage of the virtual machine.

```
x = a * b + c * d
y = a * x
z = reveal(e)
```

Figure 1: A sample program.

For way of illustration, Figure 1 shows a possible high-level program, and Figure 2 shows the corresponding output of the compiler. The semantics are as follows: Registers starting with s hold secret values, registers starting with c clear values. adds, addm, subs, mulc, and mulm denote addition, subtraction, and multiplication, respectively. The result is assigned to the first register, with the last letter of the instruction denoting the type of the inputs: s for secret, c for clear, and m for mixed. The instruction triple loads a multiplication triple into the given registers. Finally, startopen opens any number of secret registers, and stopopen assigns the results of the last startopen to the clear registers. The compiler mapped a-e to s0-s4, x to s5, y to s6, and z to c0. Note that the openings required for the multiplications a*b and c*d and the opening of e are all merged together in the first startopen/stopopen pair. Furthermore, the triple needed for the multiplication a*x is loaded only after that. This allows for the registers s6-s8 to be used for something else prior to that, thus reducing the register usage.

Vector operations.

The virtual machine also supports vector instructions like adding two vectors of registers and storing the result in a vector of regis-

```
triple s13, s10, s14
triple s11, s12, s5
subs s9, s3, s10
subs s7, s2, s13
subs s6, s1, s12
subs s8, s0, s11
startopen s8, s6, s7, s9, s4
stopopen c1, c3, c4, c5, c0
triple s7, s8, s6
subs s9, s0, s7
mulc c2, c4, c5
mulm s13, s13, c5
mulm s10, s10, c4
adds s10, s14, s10
adds s10, s10, s13
addm s10, s10, c2
mulc c2, c1, c3
mulm s11, s11, c3
mulm s12, s12, c1
adds s5, s5, s12
adds s5, s5, s11
addm s5, s5, c2
adds s5, s5, s10
subs s10, s5, s8
startopen s9, s10
stopopen c1, c2
mulm s8, s8, c1
adds s6, s6, s8
mulm s7, s7, c2
adds s6, s6, s7
mulc c1, c1, c2
addm s6, s6, c1
```

Figure 2: Compiled sample program.

ters. This does not significantly improve the performance of the runtime (in that one could obtain the same effect using multiple copies of the non-vectorized instructions), but vectorized instructions reduces memory and time needed for the compilation. Vector instructions are based on the basic instructions and take the size of the vector as an additional parameter. The further parameters have the same format as for basic instruction. Register and memory addresses are interpreted as the base address of the vector, and constants as constants. Furthermore, the high-level language allows to declare vectors of secret and clear values. Operations on those are compiled into vector instructions.

The key reason for enabling such vector instructions is to enable SIMD processing of multiple programs in the same thread. By introducing vectorized processing we can obtain high SIMD parallelism without compromising the speed of the compiler. We shall see later that the SIMD execution of higher level operations, such as floating point operations and AES evaluations, allow one to obtain a higher throughput without sacrificing latency too much. Thus the vectorized instructions give us a another way of obtaining higher throughput via parallelism, in addition to the execution of multiple tapes in multiple threads.

5. RUN TIME ARITHMETIC

The runtime comes in two variants. One for working with finite fields $\mathbb{F}_q$ where q is a "large" prime, say $q \approx 2^{128}$. This runtime variant is more suited to general purpose computations; it supports integer operations and working on fixed and floating point numbers as we shall describe later. Here we assume a statistical security parameter of 40, i.e. we allow deviations in the statistical distributions of the various shared value output of 2^{-40}. The second runtime is for working with functionalities more related to binary circuit descriptions, and it works in the finite field $\mathbb{F}_{2^{40}}$. The choice of this specific finite field is to obtain a probability of an active adversary cheating of 2^{-40}, as well as to enable the efficient evaluation of the AES functionality using the field embedding technique described in [13]. We now discuss each variant in turn:

5.1 The Large Prime Variant

Here we represent a signed t-bit integer value, where $t < 128$, by its value modulo the 128-bit prime q. Using the basic embedding of the range $[-2^{t-1}, 2^{t-1} - 1]$ into $[0, q-1]$, we can then provide standard integer operations such as addition, multiplication and comparison. Note however that "wrap around" will only occur in the range $[0, q-1]$ and not in $[-2^{t-1}, 2^{t-1} - 1]$ unless special code is produced to do so.

For some computations, we may wish to decompose a secret shared integer into bits and then perform bitwise operations on the resulting shared bits. This can be done using fairly standard bit decomposition techniques [7], but we achieve further efficiency gains through use of shared bits from the preprocessing. In particular the following bit decomposition technique is used in the floating point addition protocol of [1] which we use. We pause to overview how a bit decomposition is performed using the precomputed shared random bits: Given a shared value $\langle x \rangle$ which is known to represent an integer which is $m < 128 - 40 = 88$ bits long, we use a set of $m + 40$ shared bits $\{\langle b_i \rangle\}_{i=0}^{m+39}$ (loaded via the bit operation) to obtain t sharings of the t least significant bits of x. The shared masking value $\langle y \rangle = \sum_{i=0}^{m+39} 2^i \langle b_i \rangle$ is first computed, and applied to $\langle x \rangle$ to obtain $\langle c \rangle = 2^{m+40} + 2^m + \langle x \rangle - \langle y \rangle$. The value $\langle c \rangle$ is then opened and the result taken modulo 2^m to obtain $r = x - y \mod 2^m$. We then take the m least significant bits of r, and the shared bits $\{\langle b_i \rangle\}_{i=0}^{m-1}$, and execute the circuit for binary addition on these to compute the result. Since $\langle x \rangle$ is known to be m bits long, the length of $\langle y \rangle$ provides 40 bits of statistical security when revealing $x - y$.

This last step requires us to efficiently evaluate binary circuits for shared bit values over a large finite field. The usual approach is to execute a squaring for every XOR operation, since $a \oplus b = (a-b)^2$ over the integers for binary a and b. This is relatively expensive; so we instead adopt one of two different approaches.

- For the bitwise addition circuit used in bit decomposition above, we use the technique from [30]. In this circuit for every XOR of two bit values we also need to compute their AND. If $a, b \in \{0, 1\}$ then computing $a \wedge b$ is the same as computing their integer product; and compute $a \oplus b$ is the the integer calculation $a + b - 2 \cdot a \cdot b$, and so comes at no additional cost (since $a \wedge b$ is also needed).

- For more general binary circuits, the overhead of multiplying at every XOR gate can be costly, so we propose an alternative method for avoiding this. We simply execute an addition operation instead of XOR, and keep track of the maximum size of the resulting shared integer. When the result may exceed 2^{128-40}, or we need to return the final bit value, we can reduce the integer modulo two by applying the bit decomposition procedure (with $m = 1$) to output the least significant bit. This allows us to emulate binary circuits of reasonable depth at very little extra cost, bar the unavoidable overhead of arithmetic modulo q.

Fixed and floating point arithmetic: In addition to integer types, our compiler supports fixed and floating point arithmetic. Our fixed point arithmetic type consists of a secret shared integer of size up to 64 bits, with the point fixed at 32 binary places. Our secure floating point representation is based on the IEEE single precision format, where the significand has 24 bits of precision and the exponent 8 bits. This allows both significands and exponents to be stored as 128-bit field elements, leaving plenty of room for expansion after addition and multiplication with a 40-bit statistical security parameter.

The results of floating point operations are compliant to the IEEE standard, except we do not currently handle errors for overflow or invalid operations. Clearly any error handling performed must be secret to preserve privacy; we could use secret shared bits to handle error flags (as described in [1]) but chose to omit this for simplicity.

Note that we always use deterministic truncation in our protocols, and not the probabilistic rounding method of [8]. Whilst more efficient, the probabilistic method leads to lower accuracy and more complex analysis of errors, and is impractical for operations like comparison, where a deterministic result is required.

5.2 The $\mathbb{F}_{2^{40}}$ Variant

We also present results for a runtime based on secret sharing over the field $\mathbb{F}_{2^{40}}$. This field size is chosen to enable us to upper bound the probability of an active adversary going undetected by of 2^{-40}, which is a common value in previous work on actively secure MPC protocols. In addition the field $\mathbb{F}_{2^{40}}$ enables us to efficiently embed a copy of $\mathbb{F}_{2^8}$ within it; this was shown to be useful in [13] in evaluating an AES functionality. In a similar way we embed the binary field $\mathbb{F}_2$ into $\mathbb{F}_{2^{40}}$ thus enabling the evaluation of binary circuits.

Unlike the case of the large prime characteristic field, for the characteristic two fields, our offline phase only produces triples, bits and input sharings. We shall see in Section 6.2 that our runtime is able to evaluate the AES functionality rather efficiently. In particular one can obtain a factor of 100 improvement in either latency of throughput on the previous best runtimes for AES evaluation with active security [13]. In addition in trading throughput for latency one can always obtain a factor of 10 improvement in either of these values (and a factor of 100 improvement in the other) over the results in [13]. However, we shall see that the runtime is less efficient at evaluating complex general binary circuits.

6. EXPERIMENTAL RESULTS

Our basic experimental setup was as follows. We used two machines with Intel i7-2600S CPU's running at 2.8 GHz with 4GB of RAM, connected by a local area network. Each machine represents a party to the computation. Our experimental setup could have been performed with more than two parties with only a marginal decrease in performance.

Each program was written using our system and then compiled to bytecode tapes. Each tape was run ℓ times (with ℓ at least 50, the precise value depending on the experiment being carried out) sequentially in a single thread, and then the experiment repeated using tapes which had been vectorized with our SIMD instructions. For the SIMD experiments, a tape executed n versions of the algorithm in parallel with all of the communication batched together, for values of n in $\{1, 2, 3, 4, 5, 10, 16, 32, 64, 128\}$. Finally the same set of experiments were carried out running in t threads instead of one, for $t \in \{1, 4, 7\}$. The upper bound of seven was chosen since our processors had eight cores and one thread was acting as the controller thread.

For all experiments we measured performance using *throughput* and *latency*. Latency is simply the average execution time for $n \cdot t$ operations (obtained by dividing the total runtime by ℓ), whilst throughput is the number of operations executed per second. We plot these measures against other in the graphs that follow, showing tradeoffs that are possible depending on the application. Reading the graphs from left-to-right, the number n of operations in parallel is increased; note that the first data point always corresponds to $n = 1$, for purely sequential operations. Higher values of n result is higher latency as more data needs to be passed across the network in each communication. However, we can also see for most functionalities that at a certain point increasing n no longer buys us any extra throughput, and indeed it can decrease the throughput in some cases.

The reason for focusing on the trade-off between throughput and latency is as follows. Whilst many previous works on MPC focus on throughput only (i.e. number of AES operations per second), they often hide the high latency needed to achieve such high throughputs. In a real application one will have a specific latency one is trying to achieve, with throughput often being a secondary consideration. By presenting graphs which show the trade-off between throughput and latency, one can more accurately see the kind of performance characteristics a real world application would enjoy.

6.1 Arithmetic Circuits in Large Characteristic

In this section we detail the performance of our runtime when working in $\mathbb{F}_q$, for q a 128-bit prime. The most basic operation interactive operation is multiplication of secret shared values, for which we give timings below. We then discuss integer comparison, where we assume the input values are guaranteed to be 32 bit or 64 bit signed integers. Integer comparison is, unlike for standard processors, more expensive than multiplication. Having integer comparison allows us to implement sorting of lists with sorting networks. From comparison we move onto computing on approximations to real number computations using fixed point and floating point arithmetic. Table 6.1 summarizes the various costs and complexities of the algorithms discussed in this section.

6.1.1 Integer Multiplication

The basic non-local operation performed by the run time is executing an integer multiplication in the field $\mathbb{F}_q$. This requires a single round of communication, with each player needing to send two elements in $\mathbb{F}_q$ to each other player. In Figure 3 we present the how the latency and throughput varies depending on how many threads are used, and how many multiplications per thread are executed in a SIMD like manner per thread.

6.1.2 Integer Comparison

Comparison of secret shared integers is a fundamental operation in secure computation, used extensively throughout our fixed and floating point arithmetic below. The protocols we use all subtract one input from the other and compare the result to zero by extracting the most significant bit without doing bitwise decomposition.

For the comparison operation we implemented two protocols described by Catrina and de Hoogh [7], one with logarithmic round complexity and one in constant rounds but with slightly higher communication costs. The logarithmic rounds solution evaluates a binary circuit for computing the carry bit from binary addition, using this to obtain the comparison result. The constant rounds variant takes a different approach, using an efficient prefix multiplication protocol to compute the prefix products $p_i = \prod_{j=i}^{k}(d_j + 1) = 2^{\sum_{j=i}^{k} d_j}$, where $d_i = a_i \oplus b_i$, from which the comparison bit can be derived. The prefix multiplication by Damgård *et al.* [10] works as follows: Given non-zero inputs $x_0, x_1, x_2, \ldots$ opening the masked values $x_0 a_0^{-1}, a_0 x_1 a_1^{-1}, a_1 x_1 a_2^{-1}, \ldots$ for random non-zero $a_0, a_1, \ldots$ allows to compute $a_0 a_1, a_0 a_1 a_2, \ldots$ in a con-

Protocol	Rounds	Triples	Bits	Inverses
Multiplication	1	1	0	0
32-Bit Comparison (Constant Rounds)	4	61	143	31
32-Bit Comparison (Logarithmic Rounds)	5	52	72	0
64-Bit Comparison (Constant Rounds)	4	125	207	63
64-Bit Comparison (Logarithmic Rounds)	6	114	104	0
Fixed Point Mult.	7	63	168	0
Floating Point Mult.	15	95	218	0
Floating Point Add	47	483	2185	32
Fixed Point Comp.	10	73	159	0
Floating Point Comp.	7	124	104	0

Table 2: Round and communication complexities of the large prime field protocols.

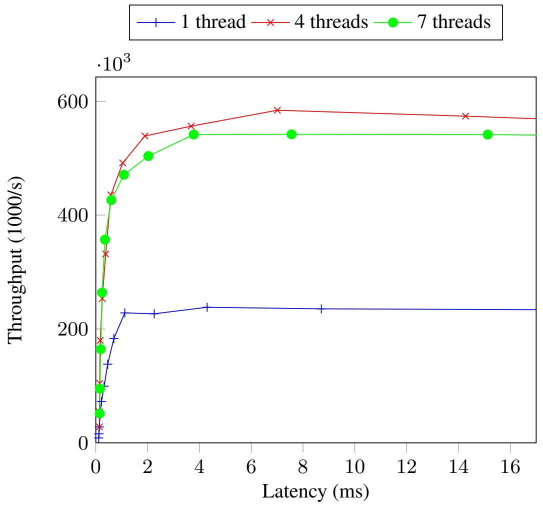

Figure 3: Integer multiplication

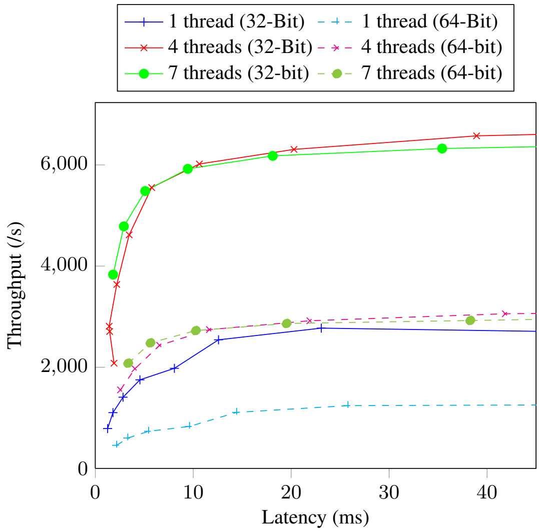

Figure 4: Integer comparison (logarithmic rounds protocol).

stant number of rounds. Catrina and de Hoogh also gave a second constant rounds protocol with better communication complexity, however this exploits properties of Shamir secret sharing which we cannot use.

For 32-bit comparison (resp. 64-bit comparison) operations with a 40-bit statistical security parameter, the logarithmic rounds solution requires six (resp. seven) rounds of communication, compared with four for the constant rounds protocol. However, the amount of data transmitted is 20% smaller; 121 (resp. 249) required openings for the logarithmic round version as opposed to 155 (resp. 315) for the constant round version (32-bit). Our experiments indicate that the saving of one round in the constant variant is dominated by the communication cost of sending more data in each round. The resulting performance is around 20% slower for the constant round method compared to the logarithmic round method. Therefore we used the logarithmic rounds protocol for all comparison and higher-level operations. Note that our implementations have slightly differing round and communication complexities to those stated in [7], firstly because we can generate shared bits without interaction using our preprocessing, and also as their constant rounds solution uses a special public output multiplication protocol, which cannot be applied to our setting.

6.1.3 Sorting

Based on integer comparison, we implemented odd-even merge sorting networks similarly to Jónsson et al. [18]. Our results are shown in Figure 5. For the implemented sizes they are similar to the results by Jónsson *et al.* Note that the figures are based on only one sequential sorting operation, using one thread for shorter lists and seven threads for longer lists. The latter improves the performance because sorting networks are highly parallelizable by design.

Sorting networks are the most efficient known method for fully private sorting because the most sorting algorithms rely on branching, which is not possible in multi-party computation if the condition is supposed to stay secret. Sorting networks only use compare-and-swap operations, which guarantee that the first output is the minimum of the two inputs. It is straightforward to implement this operation using comparison. Odd-even merge sorting networks have complexity $O(n \cdot \log^2 n)$. While this is not optimal, there is no known sorting network with complexity $O(n \cdot \log n)$ that is efficient for practical parameters.

6.1.4 Fixed and Floating Point Arithmetic

As explained earlier fixed point arithmetic is implemented using a secret shared integer of size up to 64 bits, with the point fixed at 32 binary places. Here addition is simply a local operation (as with integer arithmetic) and multiplication consists of multiplying

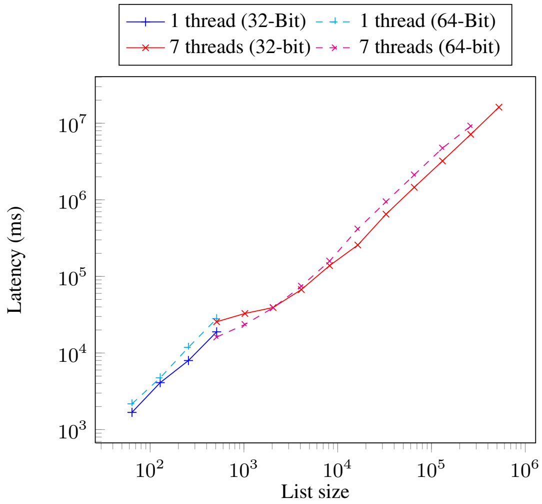

Figure 5: Sorting.

the two integers and truncating the result by 32 bits. This is essentially the same cost as a 32-bit comparison operation. The relevant performance is presented in Figures 7 and 8.

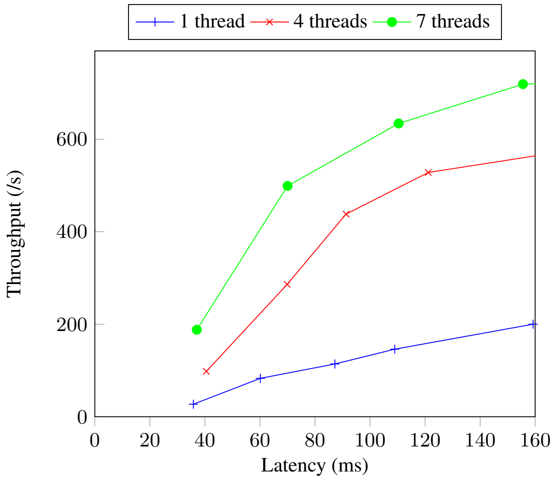

Figure 6: Floating point addition

We use the floating point protocols from [1] for addition, multiplication and comparison (see Figures 6, 7, 8). From these operations other more complex floating computations can be built up. As was also the case with integer comparison earlier, the complexities of our protocols improve upon those stated in [1] due to the preprocessing we use. Floating point protocols make extensive use of comparison and bitwise addition protocols, for which using precomputed shared bits improves performance considerably. Recall that our floating point operations are almost compliant to the IEEE standard for single precision floating point numbers; the only deviation being how we handle errors for overflow and invalid operations.

As can be seen in Figure 7, the difference in performance between fixed and floating point multiplication is not huge; floating point is only a factor of two slower. It is in addition, however, where floating point operations really suffer. This is due to the need to obliviously align the exponents of both inputs, whereas fixed point addition is a simple local operation (as with integers). With the comparison operation, floating point is slightly faster due to the smaller lengths of integers involved, despite the protocol being more complex.

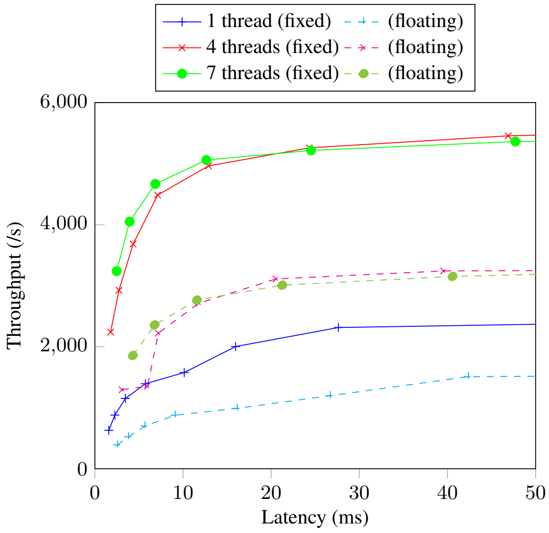

Figure 7: Fixed and floating point multiplication

Our throughput results compare favourably with those for a passively secure, three-party protocol based on Shamir secret sharing from [1]. They had a similar setup to ours, also running on a local network, and our operations are roughly an order of magnitude faster when maximum parallelism is used, probably due to our sophisticated implementation techniques described earlier.

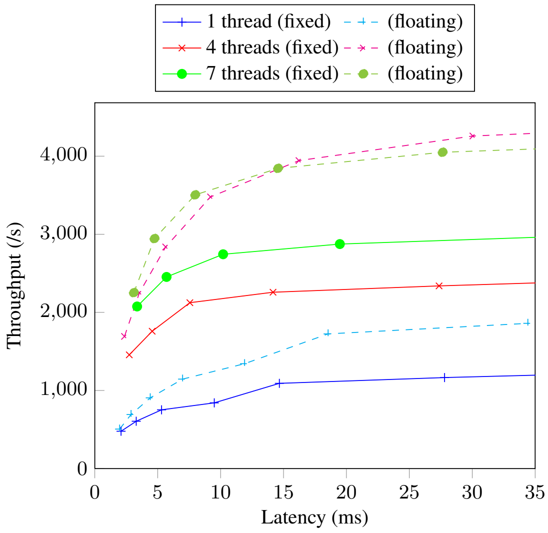

Figure 8: Fixed and floating point comparison operation (less than)

6.2 Circuits in Characteristic Two

For our experiments in characteristic two we settled on the following six functionalities as exemplars: AES (with and without the key schedule included), DES (again with and without the key schedule included), MD5, SHA-1 and SHA-256. The reason for selecting these was firstly, as has already been mentioned, AES is the standard example in the field and secondly, the circuits for DES, MD5, SHA-1 and SHA-256 are readily available. In addition if one is using AES as a prototypical PRF in an application it is interesting to compare just how much more efficient it is compared to other PRF's used in various cryptographic operations.

In Table 6.2 we present various statistics associated with these functionalities; how many multiplication triples they consume, how many bits, and how many rounds of communication are required, all for a single invocation of the functionality. Note, this latter figure is determined automatically by our compilation strategy to be the minimum necessary given the algorithm used to compute the function. The DES, MD5, SHA-1 and SHA-256 functionalities were computed via their standard binary circuit description whereas the AES functionality made use of arithmetic description over $\mathbb{F}_{2^8}$ and the efficient bit-decomposition method to compute the S-Box from [12, 13].

6.2.1 AES

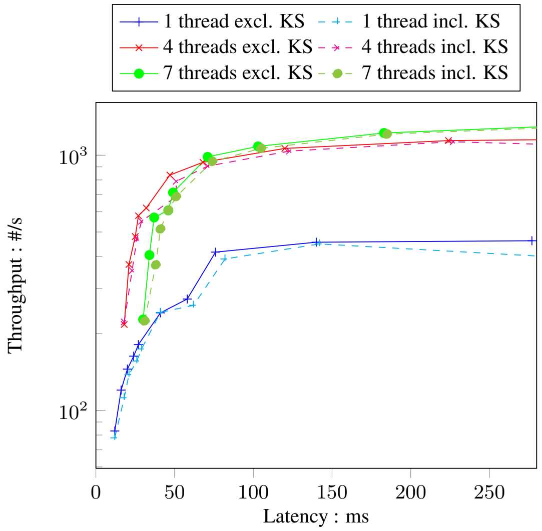

Figure 9: AES runtimes

In this functionality we assume a single AES secret key k, which has been shared between the parties. The goal of the functionality is to evaluate the AES function on a single block of public data, producing a shared output ciphertext block. From Figure 9 we can see that having a pre-expanded secret key shared between the parties makes very little difference in the execution times. We also see that with either four or seven threads a throughput of roughly 1000 blocks per second is easily obtainable; with a latency of around 100ms. The smallest latency comes when using a single thread and only batching up a single AES execution at a time; here we obtain a latency of around 12ms, but only a throughput of 83 blocks per second. Note, that the performance we obtain with our runtime is at least an order of magnitude better than that obtained in [13], without introducing any algorithmic improvements. Our improvement is purely down to a deeper understanding of how to schedule and execute MPC instructions.

6.2.2 DES

In Figure 10 we present a similar experiment for the DES cipher. Notice, just as for AES there is little difference between evaluating the cipher with a pre-expanded key and having to evaluate the key schedule using MPC. Except in the case of running seven threads, in which case having to execute the key schedule significantly reduces the throughput. The latency and throughput are about ten times worse than what was obtained from AES. This however is not (directly) because we are evaluating a binary circuit over $\mathbb{F}_2$ as opposed to working with bytes in $\mathbb{F}_{2^8}$, it is more a function of the number of rounds of communication needed and the amount of data communicated in those rounds.

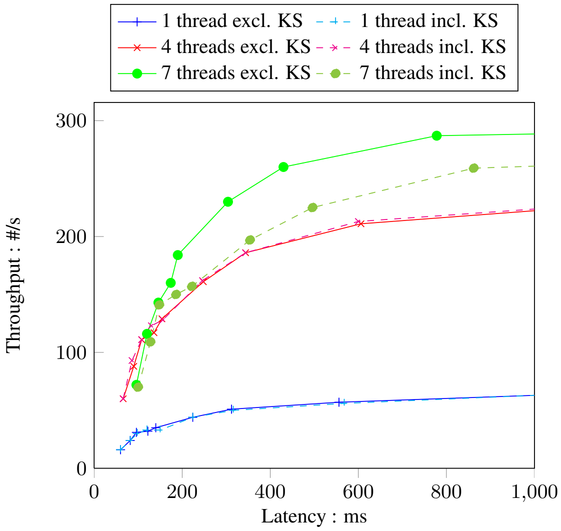

Figure 10: DES runtimes

6.2.3 MD5, SHA-1 and SHA-256

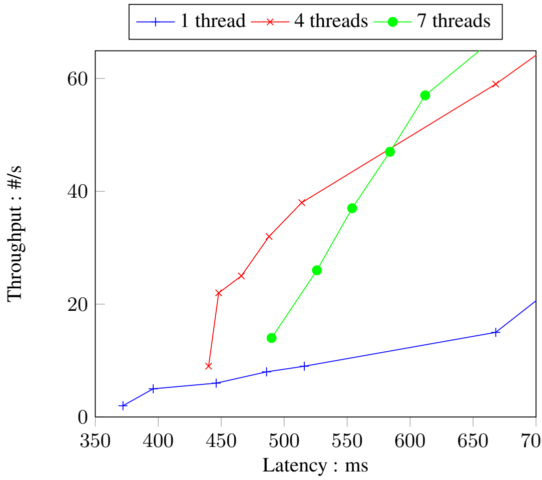

Figure 11: MD5 runtimes

When applying the runtime to the MD5, SHA-1 and SHA-256 circuits one sees that the number of multiplications/rounds has an even more dramatic effect. The latency increases and the throughput decreases as the number of rounds increases, and the number of triples consumed increases. See Figures 11, 12 and 13 for details.

7. ACKNOWLEDGEMENTS

The third author was supported in part by a Royal Society Wolfson Merit Award. This work has been supported in part by ERC

Functionality	Rounds	Triples	Bits
AES (incl. Key Schedule)	50	1200	3200
AES (excl. Key Schedule)	50	960	2560
DES (incl. Key Schedule)	263	18124	0
DES (excl. Key Schedule)	261	18175	0
MD5	2972	29084	0
SHA-1	5503	37300	0
SHA-256	3976	90825	0

Table 3: Round and required pre-processed data for the various functionalities in characteristic two.

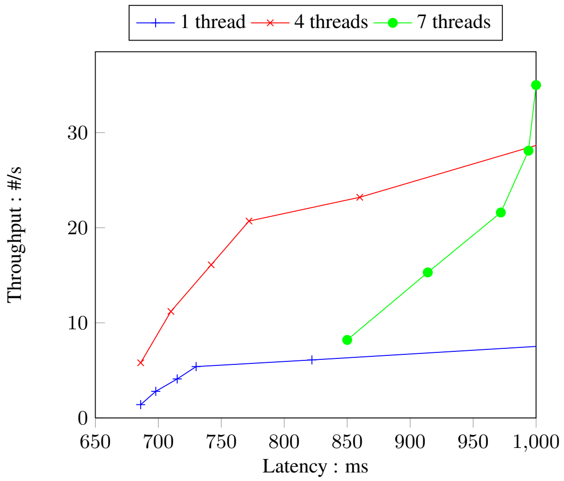

Figure 12: SHA-1 runtimes

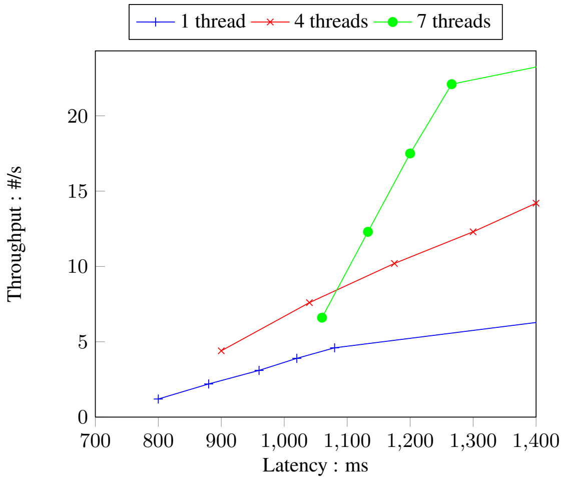

Figure 13: SHA-256 runtimes

Advanced Grant ERC-2010-AdG-267188-CRIPTO and by EPSRC via grant EP/I03126X.

8. REFERENCES

[1] M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele. Secure computation on floating point numbers. In *Network and Distributed System Security Symposium – NDSS 2013*, 2013.

[2] D. Beaver. Efficient multiparty protocols using circuit randomization. In J. Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.

[3] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In P. Ning, P. F. Syverson, and S. Jha, editors, *ACM Conference on Computer and Communications Security*, pages 257–266. ACM, 2008.

[4] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In J. Simon, editor, *STOC*, pages 1–10. ACM, 1988.

[5] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188, 2011.

[6] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[7] O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In J. A. Garay and R. D. Prisco, editors, *SCN*, volume 6280 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 2010.

[8] O. Catrina and A. Saxena. Secure computation with fixed-point numbers. In *Financial Cryptography*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2010.

[9] S. G. Choi, K.-W. Hwang, J. Katz, T. Malkin, and D. Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. In O. Dunkelman, editor, *CT-RSA*, volume 7178 of *Lecture Notes in Computer Science*, pages 416–432. Springer, 2012.

[10] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In S. Halevi and T. Rabin, editors, *Theory of Cryptography – TCC 2006*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[11] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In S. Jarecki and G. Tsudik, editors, *Public Key Cryptography – PKC 2009*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.

[12] I. Damgård and M. Keller. Secure multiparty AES. In R. Sion, editor, *Financial Cryptography*, volume 6052 of *Lecture Notes in Computer Science*, pages 367–374. Springer, 2010.

[13] I. Damgård, M. Keller, E. Larraia, C. Miles, and N. P. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In *SCN*, volume 7485 of

Lecture Notes in Computer Science, pages 241–263. Springer, 2012.

[14] I. Damgard, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical Covertly Secure MPC for Dishonest Majority – or: Breaking the SPDZ Limits. In *To appear - ESORICS*. Springer, 2013.

[15] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[16] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Tasty: tool for automating secure two-party computations. In E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 451–462. ACM, 2010.

[17] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*. USENIX Association, 2011.

[18] K. V. Jónsson, G. Kreitz, and M. Uddin. Secure multi-party sorting and applications. *IACR Cryptology ePrint Archive*, 2011:122, 2011.

[19] F. Kerschbaum. Automatically optimizing secure computation. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 703–714. ACM, 2011.

[20] B. Kreuter, A. Shelat, and C.-H. Shen. Towards billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium – 2012*, pages 285–300, 2012.

[21] J. Launchbury, I. S. Diatchki, T. DuBuisson, and A. Adams-Moran. Efficient lookup-table protocol in secure multiparty computation. In P. Thiemann and R. B. Findler, editors, *ICFP*, pages 189–200. ACM, 2012.

[22] S. Laur, R. Talviste, and J. Willemson. Aes block cipher implementation and secure database join on the sharemind secure multi-party computation framework, 2012.

[23] Y. Lindell, E. Oxman, and B. Pinkas. The IPS compiler: Optimizations, variants and concrete efficiency. In P. Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 259–276. Springer, 2011.

[24] Y. Lindell, B. Pinkas, and N. P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In R. Ostrovsky, R. D. Prisco, and I. Visconti, editors, *Security and Cryptography for Networks – SCN 2008*, volume 5229 of *Lecture Notes in Computer Science*, pages 2–20. Springer, 2008.

[25] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - Secure two-party computation system. In *USENIX Security Symposium – 2004*, pages 287–302, 2004.

[26] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700. Springer, 2012.

[27] J. B. Nielsen and C. Orlandi. LEGO for two-party secure computation. In *TCC*, volume 5444 of *Lecture Notes in Computer Science*, pages 368–386. Springer, 2009.

[28] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In M. Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009.

[29] A. Rastogi, P. Mardziel, M. Hicks, and M. A. Hammer. Knowledge inference for optimizing secure multi-party computation. Manuscript, 2013.

[30] SecureSCM Project:. Cryptographic aspects - security analysis, 2010. secureSCM deliverable D9.2.

[31] A. Shelat and C.-H. Shen. Two-output secure computation with malicious adversaries. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 386–405. Springer, 2011.

[32] SIMAP Project. SIMAP: Secure information management and processing. `http://alexandra.dk/uk/Projects/Pages/SIMAP.aspx`.

APPENDIX

A. BRANCHING/LOOPING

Due to the way the precomputed data is processed the runtime needs to know before executing a specific bytecode tape within a thread how much precomputed data will be consumed. This produces a particular issue in relation to branching and looping. We distinguish three cases:

- Loops where the number of iterations are known at compile time.
- Conditional branches forwards (as in `if-then-else` constructions).
- Conditional branches backwards (as in `do-while` loops).

We note that conditional branching can only be executed on opened values; we do not allow branching on shared values. We now discuss each of the above three cases in turn.

For loops where the number of iterations is known at compile time, we can determine the amount of precomputed data used within the loop by simply counting each instruction which loads data by the requisite number of times the loop is executed; with a suitable modification for nested loops. Forward conditional branches essentially allow one to conditionally skip code portions; thus by simply counting the total number of instructions which load precomputed data we will obtain an upper bound on the amount of precomputed data; this will be enough for the runtime to allocate different portions of the precomputed data to each bytecode tape within each thread.

The main issue occurs with conditional branches backwards; these arise for loop constructions where the number of iterations cannot be determined at compile time. Here a design decision has to be made, either we install logic to cope with this case (which would incur a large performance penalty) or we restrict the use of such constructs in some way. We decided to come up with the following constraint, after considering various use-cases. The main problem arises from not being able to allocate portions of the existing pool of precomputed data to each thread. Thus we impose a constraint on the main control thread in that it cannot execute any bytecode tape containing a conditional backward branch with any other thread at the same time. This means that tapes which could in theory be executed concurrently will be executed serially. Then the actual amount of precomputed data consumed during the execution can be determined; rather than estimated ahead of time. We found this compromise provided a suitable balance between efficiency and applicability; we can process the single tape very fast, at the expense of not being able to execute multiple tapes in parallel for such situations.