Low-communication SMC protocols from Boolean circuits

Report in MTAT.07.022 Research Seminar in Cryptography, Spring 2016

Author: Sander Siim
Supervisor: Sven Laur

May 24, 2016

Abstract

This report describes experiments in generating low-communication secure multi-party protocols on arithmetic circuits by providing a translation from a Boolean circuit format. Boolean circuits are typically used for Yao’s garbled circuits protocol, providing a constant-round computation with relatively large communication overhead. BGW-style protocols based on ring or field arithmetic have lower communication, but complex bit-level computations may be difficult and inefficient to represent with arithmetic circuits. We provide a toolchain for generating low-communication BGW-style protocols from Boolean circuits for these kinds of bit-level computations, linking together existing SMC compilers, and present our experiment results with this toolchain.
Contents

1 Introduction ........................................ 3

2 Translating Boolean circuits to arithmetic circuits ..................................... 4
   2.1 CBMC-GC Boolean circuit compiler ........................................ 4
   2.2 The Sharemind protocol language ........................................ 7
   2.3 Translation between different circuit representations .......................... 10
      2.3.1 Basic strategy ..................................................... 10
      2.3.2 Optimizations .................................................... 11

3 Comparison of different techniques ........................................ 12
   3.1 Generated circuits ..................................................... 12
   3.2 Benchmark results .................................................... 14

4 Conclusions ............................................... 15
1 Introduction

In secure multi-party computation (SMC), a number of mutually distrusting parties aim to compute a known function on their secret inputs without revealing one’s input to the other involved parties. The practical solutions for SMC today use distributed protocols involving many parties to jointly compute the desired functionality. The bottleneck for these protocols is mostly network communication, due to latency and bandwidth constraints. For a SMC protocol, one of the most important characteristics that measures its overall efficiency is the amount of communication that is performed during a single invocation of the protocol. If the communication is very large, the available network bandwidth puts a hard upper bound on the running time of the protocol, which is in fact fairly common for SMC protocols. Another aspect to consider is the round-complexity of the protocol. This measures how many rounds of communication take place during the protocol, such that later rounds have data dependencies from previous rounds, and cannot therefore be parallelized. A high round-complexity induces an overhead caused by network latency, since each round of communication is slower due to the added latency. It is more efficient to send the whole required data in a single large communication round, since then, the latency overhead is added only once. Overall, for better performance, we want protocols with low total communication and also low round-complexity.

Yao’s garbled circuits is a constant-round SMC technique, that uses a Boolean circuit representation of the function that is computed, however its total communication requirement is quite high [12, 8]. On the other hand, BGW-style protocols use arithmetic circuits over rings or fields and have linear round-complexity in multiplicative depth of circuit, but total communication size is much smaller [2]. In this sense, BGW-style protocols and Yao’s protocol have a trade-off in their network performance characteristics.

For more involved bit-level operations, arithmetic circuits that use primitive gates on longer bit-length arguments may not be optimal. On the other hand, Yao’s garbled circuits protocol uses Boolean circuits, but the communication overhead is much larger for computing a bitwise AND-gate, compared to the overhead of a BGW-style multiplication gate. The goal of this seminar report is to experiment with using BGW-style computation on a Boolean circuit, to achieve better overall communication complexity and thereby more efficient vector operations on larger inputs.

The recent domain-specific language for protocol development (PDSL) on the Sharemind secure computation platform allows to generate optimized BGW-style protocol code based on an arithmetic circuit [7]. For Yao’s protocol, optimized Boolean circuits can be generated for example using the CBMC-GC compiler for C [1, 4].

In this report, we build a translator from the CBMC-GC Boolean circuit format to the arithmetic DAG (directed acyclic graph) format of Sharemind’s PDSL. We use this translator to generate protocols for Sharemind based on the Boolean circuits generated.
by CBMC-GC. As a test of feasibility, we implement secure floating-point addition and multiplication using this approach. Floating-point arithmetic is a suitable test case, since it inherently involves bitwise operations, which are more suitably described using Boolean circuits. We compare our generated protocols to both Yao’s garbled circuits approach and Sharemind’s existing BGW-style floating point protocols developed directly using the PDSL. All protocols described in this report adopt Sharemind’s three-party security model, which provides security against passive adversaries, statically corrupting at most one of the three computing parties.

2 Translating Boolean circuits to arithmetic circuits

2.1 CBMC-GC Boolean circuit compiler

The CBMC-GC compiler is capable of translating a function described in ANSI C to a Boolean circuit representation [4]. The compiler is built upon the CBMC bounded model checker for ANSI C, which allows to statically verify assertions in a C program. Essentially, the model checker produces a boolean formula for the program, where the inputs and assertions are variables. By solving a Boolean satisfiability problem, the model checker tries to find program traces that would violate the assertions. The CBMC-GC compiler uses this Boolean representation of the program to generate the corresponding Boolean circuit.

A Boolean circuit is in essence a graph with Boolean logic gates as vertices and wires that connect the inputs and outputs of gates as edges. The circuit has a fixed number of input wires, which take 1 bit of input each, and direct these inputs to the circuit’s gates. Typical gates are AND and XOR, which take two input bits and produce an output bit corresponding to the logical operation. The circuit is then evaluated topologically, as successive gates compute the inputs for the next gates. Finally, the output is read from the circuit’s output wires. An example of a Boolean circuit is given in Figure 1.

\[
\text{AND} \quad \text{XOR} \quad \text{OR} \\
\text{u} \quad \text{v} \quad \text{w} \\
\text{NOT} \quad \text{NOT} \\
\]

Figure 1: An example of a Boolean circuit, which calculates \( w = u \geq v \) for 1-bit values. The circuit calculates the Boolean formula \( w = (u \land \neg v) \lor \neg(u \oplus v) \).

To exemplify the process of generating a Boolean circuit with CBMC-GC, we consider an example circuit for adding two 8-bit unsigned integers. Below is an example C
code that can be compiled with CBMC-GC.

Listing 1: "CBMC-GC input code for 8-bit addition"

```c
typedef unsigned char uint8_t;

void adder (uint8_t INPUT_A_x, uint8_t INPUT_B_y) {
    uint8_t OUTPUT_z = INPUT_A_x + INPUT_B_y;
}
```

In the above code, the circuit’s inputs are marked with special variable names beginning with "INPUT_", and the output is marked with a variable starting with "OUTPUT_". The output format of CBMC-GC is basically a listing of gates with pointers to the next gates where the output is directed. The output from the latest public version of CBMC-GC v9.3 is shown below.

Listing 2: "CBMC-GC 8-bit addition circuit"

```plaintext
1 AND 2 0:15:1 0:16:1 0:34:0
2 XOR 2 0:−8:0
3 XOR 2 0:2:1
4 AND 2 0:18:1
5 XOR 2 0:4:1 0:19:1
6 AND 2 0:21:1
7 XOR 2 0:6:1 0:22:1
8 AND 2 0:24:1
9 XOR 2 0:8:1 0:25:1
10 AND 2 0:27:1
11 XOR 2 0:10:1 0:28:1
12 AND 2 0:30:1
13 XOR 2 0:12:1 0:31:1
14 AND 2 0:34:1
15 XOR 2 0:14:1 0:33:1
16 XOR 2 0:14:0
17 XOR 2 0:−1:0
18 XOR 2 0:3:1
19 XOR 2 0:−7:0
20 XOR 2 0:4:0
21 XOR 2 0:5:1 0:18:0 0:20:1
22 XOR 2 0:−6:0
23 XOR 2 0:6:0
24 XOR 2 0:7:1 0:21:0 0:23:1
25 XOR 2 0:−5:0
26 XOR 2 0:8:0
27 XOR 2 0:9:1 0:24:0 0:26:1
28 XOR 2 0:−4:0
```
Each row describes a single gate in the circuit in the following format:

\[
\text{op inputs outPort:gate:inPort}
\]

where \text{op} defines the logical operation of the gate, \text{inputs} gives the number of inputs to the gate. One or more triples of the form \text{outPort:gate:inPort} define the gates that take this gate's output as input. \text{outPort} defines the output index wire of the gate (always 0 for gates with 1 output), \text{gate} gives the gate index receiving the output and \text{inPort} defines the input wire index (0 or 1 for gates with two inputs). Each gate's index is defined by the row number in the output file.

Additionally, separate files are output that define the circuit's input and output mappings. A negative index for a gate designates an output wire of the whole circuit. The input mappings for the above circuit are defined using similar syntax:

\[
\text{Listing 3: "CBMC-GC 8-bit addition circuit input mappings"}
\]

| InWire:#1  | 0:1:0 | 0:17:0 |
| InWire:#2  | 0:16:0 | 0:33:0 |
| InWire:#3  | 0:31:0 | 0:32:0 |
| InWire:#4  | 0:28:0 | 0:29:0 |
| InWire:#5  | 0:25:0 | 0:26:0 |
| InWire:#6  | 0:22:0 | 0:23:0 |
| InWire:#7  | 0:19:0 | 0:20:0 |
| InWire:#8  | 0:2:0 |
| InWire:#9  | 0:1:1 | 0:17:1 |
| InWire:#10 | 0:15:0 |
| InWire:#11 | 0:13:0 |
| InWire:#12 | 0:11:0 |
| InWire:#13 | 0:9:0  |
| InWire:#14 | 0:7:0  |
| InWire:#15 | 0:5:0  |
| InWire:#16 | 0:3:0  |

This kind of circuit can be used in Yao's garbled circuits protocol to securely evaluate the function between two or more parties. The canonical computation model for Yao's protocol is the two-party case, that is, the protocol is run between two parties (servers). Both parties can give inputs to the circuit, and they also learn the output of the evaluation of the circuit on those inputs, without learning the other party's input.
However, Yao’s protocol can also be applied in multi-party settings, such as Sharemind’s standard three-party model as shown in [9].

2.2 The Sharemind protocol language

The fastest protocols for Sharemind today are generated using a special compiler and protocol language PDSL [7]. It allows to describe BGW-style protocols in a concise declarative manner that is very similar to the mathematical protocol descriptions found in cryptography papers. Especially, all the complicated details regarding network communication are hidden away from the programmer and handled by the compiler. This allows to easily manage a large library of complex protocols. The compiler also provides automatic optimizations, which result in much more efficient protocols than could be hand-written by a programmer, as some protocols may be very complex.

The PDSL protocols are based on secret sharing and process secret-shared data to preserve the privacy of the inputs. In secret sharing, a secret is divided between a number of parties, such that each gets a share of the secret. These shares individually do not give any information about the secret, but can be later combined back to the original value. The two main secret sharing schemes used in PDSL are additive and bitwise sharing.

Additive secret sharing works over a ring $\mathbb{Z}_{2^k}$. A secret $x \in \mathbb{Z}_{2^k}$ is secret-shared by generating $n$ uniformly random and independent shares $x_1, \ldots, x_n \in \mathbb{Z}_{2^k}$, such that $\sum_{i=1}^{n} x_i = x$ using modular arithmetic in $\mathbb{Z}_{2^k}$. This is done by generating $x_1, \ldots, x_{n-1}$ uniformly randomly, and taking $x_n = x - \sum_{i=1}^{n-1} x_i$. This sharing guarantees that no proper subset of the shares gives any information about the original value in the sense that any possible value is equally likely.

Bitwise sharing is very similar to the additive one and gives the same security guarantees. Instead of sharing a larger integer, each bit is shared separately. Therefore, bitwise sharing operates on bit strings from $\mathbb{Z}_{2^k}$, which can be considered as binary representations of integers from $\mathbb{Z}_{2^k}$. For sharing a bit string $x \in \mathbb{Z}_{2^n}$ between $n$ parties, $n$ uniformly random bit strings $x_1, \ldots, x_n$ are generated such that $x = x_1 \oplus \ldots \oplus x_n$.

In their essence, BGW-style protocols that compute on such secret-shared data involve sending one-time-pad encrypted shares to other parties and then performing some local operations. After this process, the parties will hold shares of a new value whose real value is the corresponding output of a function on the input secret. Note that some operations do not require any communication between the parties. The additive scheme is homomorphic in terms of modular addition, since each party can simply add their shares to receive the share of the sum. Similarly the bitwise sharing scheme is homomorphic in terms of the bitwise XOR operation.

We do not describe here the PDSL high-level language itself, as for our purposes, we are interested only in the intermediate code format that it is compiled to. The intermediate code describes a directed acyclic graph (DAG) of the computations that

\[ x = x_1 + \ldots + x_n, \quad y = y_1 + \ldots + y_n. \]

Then

\[ z = x + y = \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} y_i = \sum_{i=1}^{n} (x_i + y_i) \]
need to be executed by the parties. The structure is very similar to a Boolean circuit. The nodes in the graph perform some arithmetic operation using inputs from previous nodes to produce an output that is propagated further. However, the operations can be on various bit-length arguments, not only single bits. In addition, as the PDSL protocols operate on secret-shared data, the protocol description must define the operations performed by each party separately. Therefore, each node in the DAG specifies a computation done by a single party. However, the input arguments for a node may originate from other parties. We explain the basic PDSL DAG structure on an example DAG, which performs a logical conjunction on secret-shared bits\(^3\). The DAG structure is given in Listing 4.

In the above, each row specifies a node in the DAG, starting with the identifier of the node (v0 to v38). Every node outputs a single value with a defined type. The value types are of the form uX,i where denotes an X-bit unsigned integer, and i denotes the computing party index, who will evaluate that node. The semantics of the different node types are:

- \( x = \text{input } i \ t \) \(- x\) is the input share for the \(i\)th input. The party and input bit-length are specified by \(t\).
- \( x = \text{and/xor } t \ y \ z \) \(- x\) is a value of type \(t\) calculated as the bitwise AND/XOR of the input values \(y\) and \(z\). \(y\) and \(z\) types should have the same bit-length as \(t\).
- \( x = \text{rnd } t \) \(- x\) is generated uniformly randomly. The party and bit-length of the value are specified by \(t\).
- \( \text{output } i \ x \) \(- x\) the value of \(x\) is the share of the \(i\)th output. The party and input bit-length are specified by the type of \(x\).

Besides and and xor there are many other logical and arithmetic operations available, so this is only a very partial specification of the DAG language. In the above DAG, we see that all values are bits, since every node has a u1 type. When the DAG is executed as a program between three parties, each party only evaluates the nodes whose types match its party index. If the arguments to a node have different party indexes, then these values have to be transferred over the network to the correct party. This is handled by the compiler. The visualization of the bit conjunction DAG is given in Figure 2.

The PDSL compiles the DAG to an LLVM bytecode program, which is directly executable from Sharemind. The LLVM code contains all the network communication and local operations. Additionally, the compiler performs structural optimizations on the DAG and reduces the amount of randomness sent over the network, by negotiating keys for a pseudo-random generator between pairs of parties \(^6\).

Also, the generated LLVM code is automatically vectorized, meaning that the protocol can be evaluated on scalar inputs or vectors of inputs of any length. Vectorized

\(^3\)For single bits, the additive and bitwise secret sharing are equivalent
Listing 4: "PDSL DAG for bit conjunction of secret-shared bits"

```
v0 = input 1 u1,1
v1 = input 1 u1,2
v2 = input 1 u1,3
v3 = rnd u1,1
v4 = rnd u1,2
v5 = rnd u1,3
v6 = xor u1,1 v0 v3
v7 = xor u1,2 v1 v4
v8 = xor u1,3 v2 v5
v9 = xor u1,1 v6 v4
v10 = xor u1,2 v7 v5
v11 = xor u1,3 v8 v3
v12 = input 2 u1,1
v13 = input 2 u1,2
v14 = input 2 u1,3
v15 = rnd u1,1
v16 = rnd u1,2
v17 = rnd u1,3
v18 = xor u1,1 v12 v15
v19 = xor u1,2 v13 v16
v20 = xor u1,3 v14 v17
v21 = xor u1,1 v18 v16
v22 = xor u1,2 v19 v17
v23 = xor u1,3 v20 v15
v24 = and u1,1 v9 v21
v25 = and u1,2 v10 v22
v26 = and u1,3 v11 v23
v27 = and u1,1 v11 v21
v28 = and u1,2 v9 v22
v29 = and u1,3 v10 v23
v30 = xor u1,1 v24 v27
v31 = xor u1,2 v25 v28
v32 = xor u1,3 v26 v29
v33 = and u1,1 v11 v22
v34 = and u1,2 v9 v23
v35 = and u1,3 v10 v21
v36 = xor u1,1 v30 v33
v37 = xor u1,2 v31 v34
v38 = xor u1,3 v32 v35
output 1 v36
output 1 v37
output 1 v38
```
operations are highly network-efficient for BGW-style protocols, since the latency overhead does not increase when evaluating the protocols on vectors. Only the sizes of the network messages increases, but the scheduling remains the same. For this reason, it is orders of magnitudes more efficient for example to multiply 1000 integers using a vectorized operation than to sequentially multiply them one-by-one, due to the higher round-complexity and added latency overhead.

![Visualization of the bit conjunction DAG.](image)

Figure 2: Visualization of the bit conjunction DAG.

2.3 Translation between different circuit representations

2.3.1 Basic strategy

Now let us consider the following task. Given a Boolean circuit, construct a protocol that securely computes the functionality specified by the circuit. One option is to use Yao’s garbled circuits protocol, which exactly achieves the goal of evaluating a Boolean circuit in an oblivious manner on different parties’ inputs. However, we are interested in constructing a BGW-style protocol instead, for its lower communication cost. This means that we assume the input and output bits are secret-shared between some parties. For simplicity, we assume $n = 3$. For secret-shared bits, the XOR operation is trivial, simply XOR the shares of the inputs locally for each party.

However, evaluating bitwise AND requires communication. Thus, the strategy for translating AND gates of the Boolean circuit is straight-forward. Simply substitute each AND gate with a full PDSL DAG for the conjunction protocol. The above DAG is exactly suited for this, as it cannot be further optimized in terms of communication by the compiler. Thus, the high-level translations to perform for each gate in the Boolean circuits are:

- **Input wires** — For each input wire with index $i$, add three nodes into the DAG:
  - `input i u1,1`
  - `input i u1,2`
  - `input i u1,3`
• **XOR gates** — For a XOR gate with inputs from gates $i, j$, find the node triples in the DAG that correspond to these gates $\{x_1, x_2, x_3\}$ and $\{y_1, y_2, y_3\}$, where $x_k$ is the $k$th party’s node corresponding to gate $i$ and $y_k$ the $k$th party’s node corresponding to gate $j$. Then add three xor nodes into the DAG:

- xor u1,1 x_1 y_1
- xor u1,2 x_2 y_2
- xor u1,3 x_3 y_3.

• **AND gates** — Similarly to the previous find the input nodes $\{x_1, x_2, x_3\}$ and $\{y_1, y_2, y_3\}$. Then compose the full bit conjunction DAG from Listing 4 such that the conjunction DAG’s input nodes are replaced with $\{x_1, x_2, x_3\}$ and $\{y_1, y_2, y_3\}$. Associate the output nodes with the index of the AND gate.

• **Output wires** — For each input wire with index $i$, find the corresponding output nodes $\{x_1, x_2, x_3\}$ and add three nodes into the DAG:

- output i x_1
- output i x_2
- output i x_3.

We implemented this transformation with a C++ program. It reads an input circuit in CBMC-GC format, parses the circuit, makes the above transformations and writes the result in PDSL DAG format to an output file.

### 2.3.2 Optimizations

This naive transformation has a relatively large computational and structural overhead due to the fact that all operations are done on bits. For example, packing together network messages from single bits is much more cumbersome than dealing with full bytes. This overhead becomes increasingly important for vectorized operations as our benchmarks also later show. The main advantage of arithmetic circuits is that they support operations on larger bit-length values. For example, instead of performing 64 XOR operations on bit values, one could perform a single XOR on 64-bit values. To make the transformation from Boolean circuit to arithmetic circuit more efficient, bit operations should be aggregated to larger bit-length operations.

Since we are interested mostly in performance of vectorized operations, we can to do this aggregation by having the DAG take $n$ sets of inputs and process these inputs using $n$-bit operations instead of 1-bit operations. For example, consider a Boolean circuit for a binary operation on two 32-bit integers. We can build the DAG such that it takes for example 8 pairs of 32-bit integers as inputs. Then, we combine the $i$th bits of the 8 inputs into a single byte for each $i \in \{1, \ldots, 32\}$. Then, we can perform operations on these aggregated bytes exactly the same way as the bit operations in the
original DAG. Later, we can extract the bits from the output bytes and reconstruct the correct 32-bit outputs. We also implemented this optimization with 8-, 16-, 32- and 64-bit operations.

3 Comparison of different techniques

To test the efficiency of the resulting protocols of this method, we implemented floating-point addition and multiplication, and performed benchmarks with both Yao’s protocol and our generated PDSL protocols with the same circuits. For Yao, we use the implementation of [9], which performs Yao’s protocol in the three-party setting on secret-shared data. We also use the same floating-point circuits as in [9], which are generated with CBMC-GC from a software floating-point library SoftFloat v2 [11]. We also compare these methods against existing Sharemind’s PDSL protocols for float addition and multiplication based on [5]. The existing PDSL protocols use a custom representation where the sign, mantissa and exponent values are additively shared. The Yao and generated PDSL protocols use bitwise-shared standard IEEE 754 floating-point representation.

3.1 Generated circuits

In Table 1 below, we list the number of gates and depth of the floating point CBMC-GC circuits. Table 2 lists the generated DAGs that are translated from the CBMC-GC circuits. The "_x8" DAGs perform the arithmetic operation on 8 pairs of inputs using 8-bit operations. Similarly, "_x64" process 64 input pairs with 64-bit operations. For the DAGs we present the total number of nodes and also the number of nodes that require exchanging messages to evaluate.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Total gates</th>
<th>AND gates</th>
<th>Total depth</th>
<th>AND-depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>float32_add</td>
<td>7052</td>
<td>5671</td>
<td>485</td>
<td>464</td>
</tr>
<tr>
<td>float32_mul</td>
<td>7701</td>
<td>5138</td>
<td>539</td>
<td>527</td>
</tr>
</tbody>
</table>

Table 1: Number of gates and depth of circuits compiled with CBMC-GC.

In Table 3 we summarize the communication costs for both Yao’s protocol and the PDSL protocols when using the circuits described above. For Yao, the total communication is the size of the garbled tables sent from one party to the other. Therefore, this is a single large round of one-way communication. Since we use the implementation of [9], these communication costs reflect both the free-XOR and garbled row reduction optimizations for Yao’s protocol. Free-XOR means that XOR-gates are garbled without communication, and the garbled row reduction reduces the number of encrypted truth table rows sent for each AND gate to 3. The implementation of [9] uses 80-bit keys, therefore, the communication cost for each AND gate is 240 bits.
Table 2: Sizes of DAGs translated from CBMC-GC Boolean circuits.

For PDSL DAGs, the compiler reports the total communication required to evaluate the DAG. However, this contains communication in both directions between all pairs of computing parties. Since the conjunction protocol is completely symmetrical for all parties, the communication between a single pair of parties is exactly one third of this number. The numbers in Table 3 reflect the communication between a single pair of parties.

<table>
<thead>
<tr>
<th>Method</th>
<th>Operation</th>
<th>Total communication</th>
<th>Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generated PDSL</td>
<td>float32_add</td>
<td>1.42 KB</td>
<td>364</td>
</tr>
<tr>
<td></td>
<td>float32_mul</td>
<td>1.28 KB</td>
<td>392</td>
</tr>
<tr>
<td>Existing PDSL</td>
<td>float32_add</td>
<td>1.23 KB</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>float32_mul</td>
<td>0.23 KB</td>
<td>26</td>
</tr>
<tr>
<td>Yao</td>
<td>float32_add</td>
<td>170 KB</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>float32_mul</td>
<td>154 KB</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Table 3: Comparison of total communication and round-complexity for Yao and PDSL protocols.

It can be seen, that Yao’s protocol, even with optimizations has a communication overhead at least two orders of magnitude greater than for the PDSL protocols. Our generated PDSL protocols have much higher round-complexity than the existing protocols. This is due to the high AND-depth of the CBMC-GC circuits. Since the CBMC-GC compiler is mainly intended for use with Yao’s protocol, it optimizes for a low number of AND gates, but does not optimize the depth of the circuit very well.

We noticed that CBMC-GC compiles addition circuits as full-adders, which have optimal AND-gate count, but have an AND-depth of 31 for 32-bit integers. Much better results for our generated protocols could be achieved when using adders with...
logarithmic depth, such as described in [10]. Since CBMC-GC uses hard-coded circuits for the most performance-critical operations in the circuit synthesis [4], the addition and multiplication subcircuits could be replaced with depth-optimized variants. A recent paper by Demmler et al. presents highly optimized Boolean circuits for secure computation, also for IEEE 754 floating-point operations [3], which shows that much more optimized Boolean circuits could be obtained for the same functionality. For showing the potential of our toolchain, we present the theoretical communication complexity if we would generate PDSL protocols from these better circuits using our toolchain in Table 4.

<table>
<thead>
<tr>
<th>Method</th>
<th>Operation</th>
<th>Total communication</th>
<th>Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theoretical generated PDSL from [3]</td>
<td>float32_add</td>
<td>0.46 KB</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>float32_mul</td>
<td>0.75 KB</td>
<td>47</td>
</tr>
<tr>
<td>Existing PDSL</td>
<td>float32_add</td>
<td>1.23 KB</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>float32_mul</td>
<td>0.23 KB</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 4: Comparison of total communication and round-complexity for Sharemind’s existing PDSL protocols and theoretical communication of protocols that could be generated with our toolchain from circuits of [3].

We can see that these circuits are almost on par with Sharemind’s existing PDSL protocols. Multiplication is still at least twice as efficient with the existing additive sharing based protocols, since it is more straight-forward to implement using the additive representation. However, the addition protocol would actually be more efficient for both total communication and round-complexity. This shows that the overall method of generating BGW-style protocols from Boolean circuits can lead to very efficient protocols. Additionally, better circuits could be generated by trying to optimize the C code compiled by CBMC-GC. Especially, by abandoning strict IEEE 754 compliance or reducing the precision of the operations, smaller circuits can be produced. Despite the large depth of our benchmark circuits, we still achieve somewhat surprisingly good performance results with our generated protocols as we discuss next.

3.2 Benchmark results

The benchmarks were performed on a cluster of three machines, with a dedicated fast 10 Gbit/s network link, 128 GB of RAM and two Intel Xeon E5-2640 v3 2.6 GHz/8GT/20M processors, meaning a total of 16 cores and 32 parallel threads with Intel Hyperthreading.

Our implementation of Yao’s protocol allows also for vectorizing computations, such that many circuits can be garbled and evaluated in parallel in separate threads. We use the maximum number of threads for the benchmark hardware, that is 32 parallel garbler and evaluator threads. This means at any given time 32 circuits are being processed
in parallel on separate inputs. If a thread finishes processing a circuit, it immediately starts processing another.

Table 5 presents the performance benchmark results on various input sizes in operations per millisecond. That is, how many amortized operations on scalars are done in a single millisecond. We chose the inputs sizes as multiples of 8 for simplicity, since the aggregated DAGs process a multiple of 8 inputs. For each operation we performed 5 iterations and present the average.

<table>
<thead>
<tr>
<th>Method</th>
<th>Operation</th>
<th>Input size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Generated PDSL</td>
<td>float32_add</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>float32_mul</td>
<td>0.3</td>
</tr>
<tr>
<td>Generated PDSL x8</td>
<td>float32_add</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>float32_mul</td>
<td>0.3</td>
</tr>
<tr>
<td>Generated PDSL x16</td>
<td>float32_add</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>float32_mul</td>
<td>-</td>
</tr>
<tr>
<td>Generated PDSL x32</td>
<td>float32_add</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>float32_mul</td>
<td>-</td>
</tr>
<tr>
<td>Generated PDSL x64</td>
<td>float32_add</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>float32_mul</td>
<td>-</td>
</tr>
<tr>
<td>Existing PDSL</td>
<td>float32_add</td>
<td>2.7</td>
</tr>
<tr>
<td></td>
<td>float32_mul</td>
<td>5.6</td>
</tr>
<tr>
<td>Yao</td>
<td>float32_add</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>float32_mul</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Table 5: Benchmark results of different protocols on vectorized inputs. Performance is presented as operations per millisecond. Best performance measurements for each protocol method and both operations are marked in bold.

For the generated PDSL protocols, we can see that aggregating bit-operations together into larger bit-length operations produces a large performance gain. The biggest difference is when aggregating bits into bytes, which is to be expected, since primitive bit-operations have about the same computational cost as byte operations. However, we get even more performance gains when aggregating into larger bit-lengths.

We can see that the generated protocols are about twice as fast as Yao’s protocol, even though Yao used up all of the hardware and used a constant 5 Gbps of bandwidth throughout computations. The PDSL protocols are evaluated on a single thread and use much less bandwidth.

# 4 Conclusions

Our experiments show the feasibility of generating efficient BGW-style protocols directly from Boolean circuits. However, the circuits produced by CBMC-GC are not
ideal for this use case, since they have a relatively large depth. Using better optimized circuits however could result in protocols with similar or even slightly better performance to Sharemind’s existing PDSL protocols.

Acknowledgment

The author of this report has received the Skype and Study IT in Estonia Master’s Scholarship for the academic year 2015/16, funded by Estonian Information Technology Foundation and Skype Technologies OÜ.

References


