Compiling Low Depth Circuits for Practical Secure Computation

Niklas Buescher[1(✉)], Andreas Holzer[2], Alina Weber[1], and Stefan Katzenbeisser[1]

[1] Technische Universität Darmstadt, Darmstadt, Germany
buescher@seceng.informatik.tu-darmstadt.de
[2] University of Toronto, Toronto, Canada

Abstract. With the rise of practical Secure Multi-party Computation (MPC) protocols, compilers have been developed that create Boolean or Arithmetic circuits for MPC from functionality descriptions in a high-level language. Previous compilers focused on the creation of size-minimal circuits. However, many MPC protocols, such as GMW and SPDZ, have a round complexity that is dependent on the circuit's depth. When deploying these protocols in real world network settings, with network latencies in the range of tens or hundreds of milliseconds, the round complexity quickly becomes a significant performance bottleneck.

In this work, we present ShallowCC, a compiler extension that creates depth minimized Boolean circuits from ANSI-C. We first introduce novel optimized building blocks that are up to 50 % shallower than previous constructions. Second, we present multiple high- and low-level depth minimization techniques and implement these in the existing CBMC-GC compiler. Our experiments show significant depth reductions over hand-optimized constructions (for some applications up to 2.5×), while maintaining a circuit size that is competitive with size-minimizing compilers. Evaluating exemplary functionalities in a GMW framework, we show that depth reductions lead to significant speed-ups in any real-world network setting. For an exemplary biometric matching application we report a 400× speed-up in comparison with a circuit generated from a size-minimizing compiler.

1 Introduction

In the thirty years since Yao's seminal paper [33], Secure Multiparty Computation (MPC) has transitioned from purely theoretic construction to a practical tool. In MPC, two or more parties jointly evaluate a function over their inputs in such a way that each party keeps its input hidden from the other parties. Thus, MPC provides a generic way to construct Privacy-Enhancing Technologies, which protect sensitive data during processing steps in untrusted environments. In the last decade, many new protocols and optimizations made MPC practical for various applications. Nevertheless, MPC is still multiple orders of magnitude slower than classic computation.

© Springer International Publishing Switzerland 2016
I. Askoxylakis et al. (Eds.): ESORICS 2016, Part II, LNCS 9879, pp. 80–98, 2016.
DOI: 10.1007/978-3-319-45741-3_5

The performance of most MPC protocols usually depends on the complexity of either a Boolean or an Arithmetic circuit representing the functionality to be computed. Unfortunately, the manual construction of efficient circuits is a complex, error-prone, and time-consuming task. Therefore, multiple compilers, for example CBMC-GC [17], Frigate [26], KSS [22], or the SecreC Compiler [4], have been developed that compile a functionality described in a high-level language into circuits satisfying the requirements of MPC protocols.

The creation of circuits from a high-level functionality shares similarities with hardware synthesis. Yet, hardware synthesis tools differ in two factors. First, no layout or space considerations have to be made when designing circuits for MPC. Second, the costs for different types of gates differ significantly. For example, in classic logic synthesis, Boolean NAND gates are favored over XOR gates due to their placement costs. However, in many MPC protocols the evaluation costs of all non-linear gates (e.g., AND, NAND and OR) are equivalent to each other, while the evaluation of linear gates (e.g., XOR) is essentially free [14]. Therefore, previous works on MPC compilers mainly focussed on producing circuits with a minimal number of non-linear gates.

Nevertheless, many practically relevant MPC protocols, such as BGW [3], GMW [14], Sharemind [4], SPDZ [9] and TinyOT [28] have a round complexity that is proportional to the circuit depth. Hence, for these MPC protocols it is crucial to also consider the circuit depth as a major optimization goal, because every layer in the circuit increases the protocol's runtime by the round trip time (RTT) between the computing parties. This is of special importance, as latency is the only computational resource that has reached its physical boundary. (For computational power and bandwidth, parallel resources can always be added.) Thus, asymptotically it is much more vital to minimize the depth of circuits, rather than speeding-up the computational efficiency. To illustrate these thoughts, the performance of a state-of-the-art implementation of the GMW protocol [14], such as ABY [11], shows that more than 10 million non-linear gates per second can be computed on a single core of a commodity CPU. At the same time, the network latency between Asia and Europe[1] is in the range of a hundred milliseconds. In this setting, the evaluation time of any circuit with less than 100,000 parallel gates per circuit level will increase by at least one order of magnitude. Therefore, it is worthwhile to investigate optimization and compilation techniques for the automatic creation of low depth Boolean circuits.

Even though this work focusses on depth-minimized Boolean circuits, MPC protocols using Arithmetic circuits or FHE schemes can also profit from the ideas presented here, as they require Boolean circuits for all control flow operations. Moreover, we note that a shallower and broader circuit allows for better parallelization in MPC protocols with constant round complexity, for example in Parallel Yao's Garbled Circuits [6].

[1] Even though, MPC is often benchmarked in a LAN setting, the WAN setting is the more natural deployment model of MPC.

Contribution. In this work, we present *ShallowCC*, a compiler that takes ANSI-C as input and automatically generates low depth Boolean circuits, optimized for MPC protocols that favor a minimal number of non-linear gates. Our approach for the generation of depth minimized circuits is threefold. First, we present and investigate minimization techniques that operate on the source code level. This involves the detection of sequential reductions, which can be regrouped in a tree based manner. We refer to reductions as the aggregation of multiple programming variables into a single result variable, e.g., minima computation over an array. We also present techniques to detect consecutive arithmetic operations, which can be instantiated more efficiently by a dedicated circuit rather than a composition of multiple individual arithmetic building blocks. Second, we present depth and size optimized constructions of major building blocks, e.g., adder and multiplexer, required for the synthesis of larger circuits. These hand-optimized building blocks have a depth that is significantly smaller than depth-minimized blocks presented in recent works [10,30]. An overview of significant improvements is given in Table 1. Third, we adapt multiple low level optimization methods that minimize circuit depth on the gate level. Finally, we contribute an implementation of our ideas as an extension to the open-source CBMC-GC compiler.

Table 1. *Depth of Building Blocks.* Comparison of the depth of the here presented building blocks with the previously known best constructions.

Operation	**Previous work** [30]	**This work**
n-bit Addition	$2\log_2(n)+1$	$\log_2(n)+1$
n-bit Multiplication	$3\log_2(n)+4$	$2\log_2(n)+3$
m:1 Multiplexer	$\log_2(m)$	$\lceil\log_2(\lceil\log_2(m+1)\rceil)\rceil$

Comparing with the hand optimized computations, e.g., computation of the Manhattan distance [10], we report depth reductions between 30 % and 60 %. Comparing with previous compilers, we report circuits, e.g., a privacy preserving biometric matching functionality, that are up 400 times shallower. Evaluating the depth minimized circuits with the GMW protocol [14], we observe speed-ups of the online protocol run time that are proportional to the depth savings, even for RRTs below 10ms. For example, we observe a speed-up of 400 times for the aforementioned biometric matching functionality.

Outline. Next, we discuss related work. An introduction into circuit design is given in Sect. 3. In Sect. 4 we present ShallowCC and its minimization techniques. An evaluation of ShallowCC is given in Sect. 5.

2 Related Work

Along with the early development of practical frameworks for MPC, circuit compilers have been developed, mainly because the manual creation of circuits for

privacy preserving applications requires expertise in hardware synthesis and can be an error prone task with circuits scaling to billions of gates. Here, we first discuss compilers for the creation of Boolean circuits mainly tailored towards Yao's Garbled Circuits, before discussing compilers for arithmetic circuits. Moreover, an overview of optimized circuit libraries is given.

Boolean circuit compilers. The development of compilers for MPC started with the Fairplay framework by Malkhi et al. [25]. Fairplay compiles a domain specific hardware description language (SFDL) into a gate list for the use in Yao's Garbled Circuits. Henecka et al. [16] presented the TASTY compiler with a domain specific language (DSL) that supports basic data types and arithmetic operations to allow the efficient combination of Garbled Circuits with additively homomorphic encryption. The PAL compiler by Mood et al. [27] also relies on Fairplay's SFDL input format, but aims at low-memory devices as the compilation target. The KSS compiler by Kreuter et al. [22] is the first compiler that shows scalability up to a billion of gates. KSS compiles circuits from a domain specific hardware language and employs advanced optimization methods, e.g., constant propagation or dead gate elimination. ObliVM by Liu et al. [23] is a framework for Java that enables the automatized combination of oblivious data structures with MPC. Songhori et al. [31] presented Tiny Garble, which uses commercial hardware synthesis tools to compile circuits from VHDL. One the one hand, this approach allows to use a broad range of existing functionalities in hardware synthesis, but also shows the least degree of abstraction, by requiring the developer to have experience in hardware design. Zahur and Evans [35] presented a compilation approach, named Obliv-C, that compiles a DSL into executable C code, thus, combining compiler and execution environment. Very recently, Mood et al. [26] presented the Frigate compiler, which aims at very fast and extensively tested compilation of another DSL.

The CBMC-GC compiler by Holzer et al. [17] is the first compiler that creates Boolean circuits for MPC from ANSI-C. CBMC-GC utilizes the Bounded Model Checker CBMC, originally used for the verification of C code, to reliably compile a large subset of C to circuits. ParCC, presented by Buescher et al. [6], is a source-to-source compiler, which extends CBMC-GC by the capability to compile parallel circuits. The PCF compiler by Kreuter et al. [21] is compiles C using the intermediate representation of the portable LCC compiler.

Mood et al. [26] give an overview on many of the aforementioned compilers and benchmark their performance. The authors indicate limited robustness of many existing compilers, as most have been developed for research purposes. We observe that all compilers apply various optimization methods, yet all aim at the creation of size and not depth minimal circuits.

Arithmetic circuit compilers. Multiple compilers that aim at the creation of circuits for use in secret sharing based MPC have been developed. Early compilers are the FairplayMP compiler by Ben-David et al. [2] and the VIFF compiler by Damgard et al. [8], which both compile a DSL. The Sharemind framework by

Bogdanov et al. [4] is nowadays the most advanced compiler for MPC. It compiles a DSL, implements a broad range of functionalities and supports multiple MPC protocols during runtime. The Picco compiler by Zhang et al. [36] compiles ANSI-C into interpretable arithmetic circuits, yet has not been open sourced.

Optimized circuit libraries. Kolesnikov and Schneider [19,20] presented first size optimized low-level building blocks, e.g., adder and multiplexer, for their use in Yao's Garbled Circuits. Zahur and Evans [34] presented optimized circuit structures for more advanced building blocks, such as stacks and queues. Schneider and Zohner [30] identified the need of low depth circuits for a fair comparison between GMW und Yao's Garbled Circuits and presented multiple depth minimized building blocks. Most recently, Demmler et al. [10] presented a library of low depth circuits exported from a commercial hardware synthesis tool. In this work, we compare our results with these hand optimized circuits.

3 Preliminaries in Digital Circuit Design for MPC

Digital circuit design, also known as logic synthesis, deals with the construction and optimization of digital circuits. Common optimization goals are the reduction of the placement costs and the signal delay under several physical constraints. In circuit design for MPC, however, many of the classical design criteria (e.g., signal amplification) can be omitted, because the created circuits are evaluated 'virtually' in software. In this work, we investigate the creation of Boolean circuits based on gates with two input wires, as these provide the most general circuit description. In the following paragraphs, we describe the used notation, as well as some basic concepts applied in logic synthesis.

Notation. We use s^{nX} to notate the total number of non-linear gates of a circuit, also referred to as size, and d^{nX} to denote the circuit's depth in the number of non-linear gates. Furthermore, we denote bit strings in capital letters, e.g. X, and denote their negation with $\overline{X}$. We refer to single bit at position i within a bit string with X_i. The Least-Significant Bit (LSB) is X_0. Moreover, we denote the Boolean XOR gate with $\oplus$, AND with $\cdot$ and OR with $+$. When useful, we abbreviate the AND gate $A \cdot B$ with AB.

Half- and Full-Adder. Arithmetic building blocks are constructed of smaller building blocks, namely Half-Adders (HA) and Full-Adders (FA). A Half-Adder is a combinatorial circuit that takes two bits A and B and computes their sum $S = A \oplus B$ and carry bit $C_{out} = A \cdot B$. A Full-Adder allows an additional carry-in bit C_{in} as input. The sum is computed by XOR-ing all inputs $S = A \oplus B \oplus C_{in}$, the carry-out bit can be computed by $C_{out} = (A \oplus C_{in})(B \oplus C_{in}) \oplus C_{in}$ [20]. Both, the HA and FA have size $s^{nX} = 1$ and depth $d^{nX} = 1$.

Carry-Save Adder. In the early 1960s [12], Carry-Save Addition was introduced to compute the sum of k numbers in logarithmic depth. The main component of a Carry-Save Addition is the 3:2 Carry-Save Adder (CSA). A 3:2 CSA reduces the sum of three numbers $A+B+C$ to the sum of two numbers $X+Y$ in small constant depth [29]. A CSA for three n-bit values A, B and C can be instantiated by n parallel FAs. This instantiation has a depth of one and allows to compute the partial sums X and Y for k numbers with depth $d^{nX}_{CSA}(k) = \lceil \log_2(k) - 1 \rceil$ [30].

Parallel Prefix Circuit. A parallel prefix circuit is used in depth minimizing adders and computes n outputs $O_1, \ldots, O_n$ from n inputs $X_1, \ldots, X_n$ for an arbitrary associative two-input operator $\circ$ as follows [15]:

$$O_1 = X_1, \quad O_2 = X_1 \circ X_2, \quad \ldots, \quad O_n = X_1 \circ X_2 \cdots \circ X_n.$$

All outputs can then be computed with at most logarithmic depth when applying the operator $\circ$ in a tree structure over all inputs, e.g., $O_4 = (X_1 \circ X_2) \circ (X_3 \circ X_4)$.

Two's complement. The two's complement is the common representation of signed numbers in hardware. In the two's complement, negative numbers are represented by flipping all bits and adding one. In the following sections, we assume a two's complement representation, when referring to negative numbers.

4 Creation of Low Depth Circuits

In this section, we present the design of our compiler extension ShallowCC as well as multiple depth minimization techniques. ShallowCC is built on top of CBMC-GC, which, even though being the first compiler for ANSI-C, creates circuits that are still competitive in size [26]. CBMC-GC is open sourced, well documented and shows great reliability due to is origin in model checking. Moreover, it implements powerful minimization techniques on the gate level that make it an optimal candidate to implement the ideas presented in this section.

ShallowCC follows CBMC-GC's compilation approach and adopts them for depth minimization as illustrated in Fig. 1. Adaptations and extensions are marked in gray. The compiler reads ANSI-C code with a special naming convention for input arguments and output variables (see [17] for code examples). First,

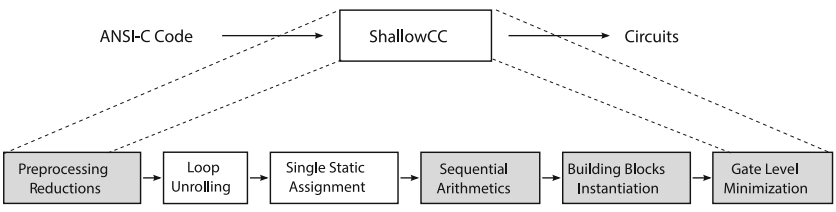


Fig. 1. ShallowCC's compilation chain from ANSI-C to Boolean circuits.

```
unsigned max_abs(int a[], unsigned len) {
  unsigned i, max = abs(a[0]);
  for(i = 1; i < len; i++)
    if(abs(a[i]) > max)
      max = abs(a[i]);
  return max;
}
```

Listing 1. Exemplary function that computes the maximum norm.

the code is preprocessed to detect and transform reduction statements on the source code level (see Sect. 4.1). In the second and third step all bounded loops and recursions are unrolled using symbolic execution and the resulting code is transformed into Single Static Assignment (SSA) form. In the fourth step, the SSA form is used to detect and annotate successive composition of arithmetic statements (see Sect. 4.1). Afterwards, all statements are instantiated with hand-optimized building blocks (see Sect. 4.2), before a final gate-level minimization takes place (see Sect. 4.3).

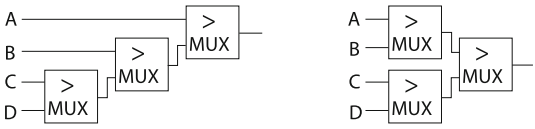
4.1 Code Level Minimization Techniques

In the following paragraphs we discuss two techniques that operate on the source code level to decrease the circuit depth.

Reduction Statements. We refer to a reduction as the compression of multiple programming variables into a single result variable, e.g., the sum of an array. Consider the code example in Listing 1. This code computes the maximum norm of a vector. It iterates over an integer array, computes the absolute value of every element and then reduces all elements to a single value, namely their maximum. A straight forward translation of the maximum computation leads to a circuit consisting of `len` -1 sequentially aligned comparators and multiplexers, as illustrated in Fig. 2a. However, the same functionality can be implemented with logarithmic depth when using a tree structure, as illustrated in Fig. 2b. Thus, when optimizing circuits for depth, it is worthwhile to rewrite sequential reductions. To relieve the programmer from this task, ShallowCC automatically replaces sequential reductions found in loop statements by tree-based reductions.

Since detecting reductions in loop statements is a common task in automatized parallelization, we adapt the recent work on parallel circuits by Buescher and Katzenbeisser [6]. The authors use the parallelization framework Par4all [1] to detect parallelism on source code level. As a side product, Par4all also identifies and annotates sequential reductions. We extend the techniques presented in [6] to parse these reduction annotations and to rewrite the code during the preprocessing phase with *clang* (source-to-source compilation). For this, we first identify the loop range and reduced variable to instantiate a code template that computes the reduction in a tree structure. This optimization improves the depth of reductions over m elements from $O(m)$ to $O(\log m)$. To give an example, for

a minimum computation of a 32-bit integer array with 100 elements, we observe a depth reduction from 592 to 42 non-linear gates, cf. Sect. 5.3.



(a) Sequential circuit (b) Tree circuit

Fig. 2. Maximimum search circuit, consisting of comparators and multiplexers.

Carry-Save Networks (CSNs). CSNs are efficient circuit constructions for multiple successive arithmetic operations that outperform their individual composition in size and depth. Consider the following lines of code as an example:

```
unsigned a, b, c, d;
unsigned t = a + b;
unsigned sum = t + c + d;
```

A straight forward compilation, as in CBMC-GC, leads to a circuit consisting of three binary adders: `sum = ADD(ADD(ADD(a,b), c), d)`. However, if it is possible to identify that a sum of four independent operands is computed, a CSA with four inputs can be initiated instead: `sum = CSA(a,b,c,d)`. This reduces the circuit's depth in this example from 18 to 7 non-linear gates.

Detecting these operations on the gate level is feasible, for example with the help of pattern matching, yet impractically costly considering that circuits reach sizes in the range of billions of gates. Therefore, ShallowCC aims at detecting these successive statements before their translation to the gate level. We do this by utilizing the capabilities of the bounded model checker CBMC [7] that ShallowCC is built upon. CBMC compiles C code into the SSA form, where each variable is written only once. The SSA form allows efficient data flow analyses and as such, also the search for successive arithmetic operations. Our detection algorithm consists of two parts. First, a breadth-first search from output to input variables is initiated. Whenever an arithmetic assignment is found, a second backtracking algorithm is initiated to identify all preceding (possibly nested) arithmetic operations. This second algorithm stops whenever a guarded or non-arithmetic statement is found. Once all preceding inputs are identified, the initial assignment can be replaced by a CSN. After every replacement, the search algorithm continues its search towards the input variables. We note that this greedy replacement approach is depth minimizing, yet not necessarily size optimal, since intermediate results in nested statements may be computed multiple times. A trade-off between size and depth is possible by only instantiating CSNs for non-nested arithmetic statements.

Quantifying the improvements, assuming that the addition of two numbers requires a circuit of depth d_{Add}^{nX}, we observe that by sequential composition $m > 2$

numbers can be added with depth $(m-1) \cdot d_{Add}^{nX}$. When using a tree-based structure the same sum can be computed with a depth of $\lceil \log_2(m) \rceil \cdot d_{Add}^{nX}$. However, when using a CSA, m numbers can be added with a depth of only $\lceil \log_2(m) - 1 \rceil + d_{Add}^{nX}$. Furthermore, multiplications and additions can be merged in a single CSN, as every multiplication internally consists of additions of partial products, cf. Sect. 4.2. For the exemplary computation of a 5×5 matrix multiplication, we observe an improvement in depth of more than 60 %, cf. Sect. 5.3.

4.2 Optimized Building Blocks

Optimized building blocks are an essential part when designing complex circuits. They facilitate efficient compilation, as they can be highly optimized once and subsequently instantiated at practically no cost during compilation. In the following paragraphs, we present new depth and size optimized building blocks constructed from Boolean gates for basic arithmetic and control flow operations.

Adder. An n-bit *adder* takes two bit strings A and B of length n, representing two (signed) integers, as input and returns their sum as an output bit string S of length $n+1$. The standard adder is the Ripple Carry Adder (RCA) that consists of a successive composition of n FAs. This leads to a linear circuit size and depth $s_{RCA}^{nX} = d_{RCA}^{nX} = O(n)$. Parallel Prefix Adders (PPAs) are widely used in logic synthesis to achieve faster addition under size trade-offs by using a tree based prefix network with logarithmic depth. PPAs have been investigated for their use in MPC [10,30]. Surprisingly, and to the best of our knowledge, none of these constructions challenged the textbook design of PPAs, which never considered the 'free' XOR cost model. In the full version of this paper[2], we prove that it is possible to replace one of the two non-linear gates by an XOR gate in every layer of the a PPA. Applying this design to the Sklansky adder, which shows the least depths of all PPAs (see taxonomy of Harris [15]), we achieve a construction with a depth of $d_{Sk}^{nX}(n) = \lceil \log_2(n) \rceil + 1$ and a size of $s_{Sk}^{nX} = n\lceil \log_2(n) \rceil$ for an input bit length of n and output bit length of $n+1$. In Table 2 a depth and size comparison of the standard Ripple-Carry adder, the Ladner-Fischer adder, as proposed in [30], the here optimized Sklansky adder, and an alternative to the Sklansky adder, namely the Brent-Kung adder [15] is given for different bit-widths. We observe that the RCA provides the least size and the Sklansky adder the least depth. The Brent-Kung adder provides a trade-off between size and depth. Both of our optimized constructions significantly outperform the previous best known depth-minimized construction in size and depth.

Subtractor. A subtractor can be implemented with one additional non-linear gate by using the two's complement representation $a - b = a + \overline{b} + 1$, with $\overline{b}$ being the negated binary representation [19]. The addition of negative numbers in the Two's complement is equivalent to an addition of positive numbers. Hence, the subtractor profits to the same degree from the optimized addition.

[2] Full version available at http://www.seceng.de/people/buescher/.

Table 2. *Adders.* Comparison of circuit size s^{nX} and depth d^{nX} of the standard RCA, the previously best known depth-optimized adder [30] and our newly optimized Brent-Kung and Sklansky adder.

	depth d^{nX}				**size** s^{nX}			
Bit-width	n	16	32	64	n	16	32	64
Ripple-Carry	$n-1$	15	31	63	$n-1$	15	31	63
Ladner-Fischer [10,30]	$2\lceil\log(n)\rceil+1$	9	11	13	$1.25n\lceil\log(n)\rceil+2n$	113	241	577
Brent-Kung-opt	$2\lceil\log(n)\rceil-1$	7	9	11	$3n$	48	96	192
Sklansky-opt	$\lceil\log(n)\rceil+1$	5	6	7	$n\lceil\log(n)\rceil$	64	160	384

Multiplier. A multiplier takes two input strings of length n as input and returns their product in form of an output bit string of length $2n$. The standard approach for multipliers is the 'school' method. Here n partial products of length n are computed and then added. This approach leads to a quadratic size $s^{nX}_{MUL,s} = 2n^2 - n$ and linear depth $d^{nX}_{MUL,s} = 2n-1$, cf. [30].

A faster addition of the partial products can be achieved when using Carry-Save Adders (CSAs, cf. Sect. 3). Such a tree based multiplier consists of three steps: First, the computation of all $n \times n$ partial products, then their aggregation in a tree structure using CSAs, before the final sum is computed using a two-input adder. The first step is computed with a constant depth of $d^{nX}_{PP} = 1$, as only one single AND gate is required. For the last step, two bit strings of length $2n-1$ have to be added. Using our Sklanksy adder, this addition can be realized in $d^{nX}_{Sk}(n) = \lceil\log_2(2n-1)\rceil+1$. The second phase allows many different designs, as the CSAs can arbitrarily be composed. The fastest composition is the Wallace tree [32], which leads to a depth of $d^{nX}_{CSA}(n) = \log_2(n)$ for MPC. Combing all three steps, a multiplication can be realized with a depth of $d^{nX}_{Wa}(n) = d^{nX}_{PP} + d_{CSA}(n) + d^{nX}_{Sk}(2n-1) = 2\log_2 n + 3$.

In Table 3 we present a comparison of the multipliers discussed above with the depth optimized one presented in [30]. Compared with this implementation, we are able reduce the depth by at least a third for any bit-width.

Table 3. *Multipliers.* Comparison of circuit depth d and size s of the school method, the multiplier given in [30] and our optimized Wallace construction.

	depth d^{nX}				**size** s^{nX}			
Bit-width	n	16	32	64	n	16	32	64
Standard	$2n-1$	45	93	189	n^2-n	496	2016	8128
MulCSA [30]	$3\lceil\log_2(n)\rceil+4$	16	19	22	$\approx 2n^2 + 1.25n\log_2(n)$	578	2218	8610
Wallace-opt	$2\lceil\log_2(n)\rceil+3$	11	13	15	$\approx 2n^2 + n\log_2(n)$	512	2058	8226

Multiplexer. A multiplexer (MUX) is the most important building block for the control and data flow of any MPC application. MUXs are used to represent conditionals and dynamic array access. A 2:1 n-bit MUX consists of two input bit strings D^0 and D^1 of length n and a control input bit C. The control input decides which of the two input bit strings is propagated to the output bit string O of the same bit length. Kolesnikov and Schneider [20] presented a construction of a 2:1 MUX that only requires one single non-linear gate for every pair of input bits by computing the output as $O = (D^0 \oplus D^1)C \oplus D^0$. This leads to a circuit size of $s^{nX}_{MUX}(n) = n$ and depth of $d^{nX}_{MUX}(n) = 1$. A 2:1 MUX can be extended to a m:1 MUX that selects between m input strings $D^0, D^1, \ldots, D^m$ using $\log_2(m)$ control bits $C{=}C_0, C_1, \ldots C_{\log(m)}$ by tree based composition of 2:1 MUXs leading to a circuit of size $s^{nX}_{MUX\_tree}(m,n) = (m-1) \cdot s^{nX}_{MUX}(n)$ with logarithmic depth $d^{nX}_{MUX\_tree}(m,n) = \log_2(m)$ [30].

We propose a further depth reduction by a logarithmic factor when constructing the multiplexer in disjunctive normal form (DNF) over all combinations of choices. Every conjunction of the DNF encodes a single choice together with the associated data wire. For MPC, this construction leads to a very low depth, because the disjunctive ORs can be replaced by XORs, as all choices are mutually exclusive. For example, a 4:1 MUX is constructed by:

$$O = D^0\overline{C_0}\,\overline{C_1} \oplus D^1\overline{C_0}C_1 \oplus D^2C_0\overline{C_1} \oplus D^3C_0C_1.$$

Thus, the depth of a m:1 MUX can be reduced to the depth of one conjunction $d^{nX}_{MUX\_DNFd}(m,n) = \lceil \log_2(\lceil \log_2(m) \rceil + 1) \rceil$. Unfortunately, a naïve implementation of a n-bit m:1 MUX_{DNFd}, as described above, increases the size to $s^{nX}_{MUX\_DNFd}(m,n) = mn \cdot \log(m)$. Since this size increase can be quite significant for larger m, we propose a second construction, referred to as MUX_{DNFs}. The idea is first to compute every choice conjunction, before AND-gating them with the data inputs, leading to a depth of $d^{nX}_{MUX\_DNFs}(m,n) = \lceil \log_2(\lceil \log_2(m) \rceil) \rceil + 1$. Now, every conjunction can be computed size efficiently, by avoiding the duplicated computations of choice combinations, e.g., the choices $C_0C_1C_2$ and $\overline{C_0}C_1C_2$ require both the computation of C_1C_2, which can be merged. This reduces the size to:

$$s^{nX}_{MUX\_DNFs}(m,n) = m + \frac{m}{2^0} + \frac{m}{2^1} + \cdots + \frac{m}{2^{m-2}} + mn < 2m + mn.$$

In Table 4 a comparison of the three MUXs is given for a different number of inputs m and a typical bit-width of 32 bits. In summary, we improved the depth of MUXs by a logarithmic factor with a moderate increase in size.

4.3 Gate Level Minimization Techniques

Minimizing the circuit on the gate level is the last step in ShallowCC's compilation chain. We first give a high level description of CBMC-GC's optimization flow, before discussing the adaptations made for ShallowCC.

Table 4. *Multiplexers.* Exemplary comparison of circuit depth d and size s a of m:1 multiplexers for a different number of inputs m of bit-width $n = 32$.

	depth d^{nX}				**size** s^{nX}			
Input choices	m	8	128	1024	m	8	128	1024
MUX_{Tree}	$\lceil \log(m) \rceil$	3	7	10	$(m-1) \cdot n$	244	4,064	31,968
$\mathbf{MUX}_{DNFd}$	$\lceil \log_2(\lceil \log_2(m)+1 \rceil) \rceil$	2	3	4	$mn \cdot \lceil \log(m) \rceil$	768	28,672	320,000
$\mathbf{MUX}_{DNFs}$	$\lceil \log_2(\lceil \log_2(m) \rceil) \rceil + 1$	3	4	5	$2m + mn$	272	1,088	34,000

Finding a minimal circuit for a given functionality is known to be Σ_2^P complete [5]. Therefore, CBMC-GC follows an heuristic approach when minimizing circuits: First, structural hashing is applied to identify and remove duplicated sub circuits. Then, a fixed-point optimization algorithm is initiated (the algorithm runs until no further improvements are made), which itself consists of two alternating phases. In the first phase, a template based circuit rewriting is executed, which applies Boolean theorems to reduce the circuit size. For example, the Idempotent law $X + X = X$ forms a template, namely an OR gate with the same inputs can safely be removed. In the second phase SAT sweeping is applied, which identifies unused gates with the help of a SAT solver. For ShallowCC, we left the structural hashing and SAT sweeping unmodified, as both help to reduce the circuit complexity. Instead, we adapt the template based rewriting phase.

The circuit rewriting in CBMC-GC only considers patterns that are size decreasing and have a depth of at most two binary gates. For depth reduction, as required in ShallowCC, however, it is useful to also consider deeper circuit structures, as well as patterns that are size preserving but depth decreasing. For example, sequential structures, $X = A + (B + (C + (D + E))$ can be replaced by tree based structures $X = ((A + B) + C) + (D + E)$ with no change in circuit size. Therefore, in ShallowCC we extend the rewriting phase by several depth minimizing patterns, which are not necessarily size decreasing. In total 21 patterns changed, resulting in more than 70 patterns that are searched for (see full version of this paper for a list of example patterns). Furthermore, we extend the formerly fixed-depth pattern matching algorithm by a recursive search to deeper sequential structures, as in the example above. To apply the new patterns in an efficient manner, we modify CBMC-GC's fixed point algorithm such that the algorithm only terminates if no further size *and depth* improvements are made or a user defined time limit is reached. Moreover, for performance reasons, the rewriting first only applies fixed depth patterns, before applying the search for deeper sequential structures.

Quantifying the improvements of individual patterns is almost impossible. This is because the heuristic approach commonly allows multiple patterns to be applied at the same time and every replacement has an influence on future applicability of further patterns. Nevertheless, the whole set of patterns that we identified is very effective, as circuits before and after gate level minimization differ up to a factor of 20× in depth, cf. Sect. 5.3.

5 Evaluation

The evaluation of ShallowCC in split in three parts. First, we compare ShallowCC with existing depth and size minimized circuits from recent works. Then, we exemplarily evaluate the different optimization techniques of ShallowCC to illustrate their effectiveness. Finally, we show that the depth minimized circuits, even under size trade-offs, significantly reduce the online time of the GMW protocol for different network configurations. We begin with a discussion of the benchmarked functionalities.

5.1 Functionalities

For comparison purposes, we focus on functionalities that have been used before to benchmark MPC. The evaluated functionalities include basic building blocks as well as more complex applications, such as biometric matching.

Arithmetic building blocks and floating point operations. Due to their importance in almost every computational problem, we benchmark arithmetic building blocks individually. For multiplication we follow the example of [26] and distinguish results for output bit strings of length n and of length $2n$ (overflow free) for n-bit input strings. Floating point calculations are necessary for all applications where numerical precision is required, e.g., privacy preserving statistics. We abstain from implementing hand-optimizing floating point circuits, but instead rely on ShallowCC's capabilities to compile a IEEE-754 compliment software floating point implementation written in C.

Distances. Various distances are used in privacy preserving protocols. The *Hamming* distance between two bit strings is the number of pairwise differences in every bit position. Due to its application in biometrics, the Hamming distance has often been used for benchmarking MPC compilers, e.g., [17,22,26,31]. The Hamming distance can be parametrized by the bit length of the input strings. The *Manhattan* distance $dist_M = |x_1 - x_2| + |y_1 - y_2|$ between to points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the distance along a two dimensional space, when only allowing horizontal or vertical moves. The *Euclidian* distance between two points is defined as $dist_E = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Due to the complexity of the square root function, it is common in MPC to benchmark the squared Euclidian distance [30].

Matrix-vector/matrix multiplication. Algebraic operations such a matrix multiplications are building blocks for many privacy-preserving applications and have repeatedly been used before to benchmark MPC [10,17,21]. Being a purely arithmetic task, its a good showcase to illustrate the automatic translation of arithmetic operations into CSNs with very low depth.

Oblivious arrays. Oblivious data structures are a major building block for the implementation of privacy preserving algorithms. The most general data structure is the oblivious array that hides the accessed index. Here, we only benchmark the array read operation, as its circuit is more complex and thus, interesting than the write operation [18].

Biometric matching. In biometric matching a party matches one biometric sample against the other's party database of biometric templates. Example scenarios are face-recognition or fingerprint-matching [13]. One of the main concepts is the computation of a distance, e.g., Euclidean, between the sample and all database entries. Once all distances have been computed, the minimal distance determines the best match. For the following experiments, we fix the dimension of a sample to $d = 4$, as it has been used before in MPC benchmarking [6,11].

5.2 Circuit Comparison

We implemented all the aforementioned functionalities in C and compiled them with ShallowCC on an Intel Xeon E5-2620-v2 CPU with a minimization time limit of 10 min. To illustrate the used sources codes, we refer the reader to the full version of this paper. The resulting circuit dimensions for different parameters and bit-widths are given in Table 5. Furthermore, the circuit size, when compiled with the size minimizing Frigate compiler and CBMC-GC v0.93 is given, as well as a comparison with the depth-minimized circuit constructions of [10,30]. The results for Frigate, [10,30] are taken from the publications.

Comparing the depth of the circuits compiled by ShallowCC with the hand minimized circuits of [10,30] we observe a depth reduction at least 30 % for most functionalities. The only exception are the floating point operations, which do not reach the same depth as given in [10]. This is because floating point operations mostly consist of bit operations, which can significantly be hand optimized on a gate level, but are hard to optimize when complied from a high-level implementation in C. When comparing circuit sizes, we observe that ShallowCC is compiling circuits that are competitive in size to the circuits compiled from the size minimizing compilers. A negative exception is the addition, which shows a significant trade off between depth and size. However, the instantiation of CSNs allows ShallowCC to compensate these trade-offs in applications with multiple additions, e.g., the matrix multiplication. In Sect. 5.4 we analyze these trade-offs in more detail. In summary, ShallowCC is compiling ANSI-C code to Boolean circuits that outperform hand crafted circuits in depth, with moderate increases in size.

5.3 Evaluation of the Optimizations Techniques

In Table 6 an evaluation of the different optimization techniques for various example functionalities is given. For every functionality the same source code is compiled twice, once with the specified optimization technique enabled and once without. Obviously, not all optimizations apply to all functionalities, therefore, we only investigate a selection of functionalities that profit from the different optimization

Table 5. Comparison of circuit size s^{nX} and depth d^{nX} compiled by the size minimizing Frigate [26], CBMC-GC v0.93 [17] compiler, the best, manually depth minimized circuits given in [10,30] and the circuits compiled by ShallowCC. Improvements are computed in comparison with the previous work [10,30]. The '-' indicates that no results were given. Marked in bold face are cases with significant depth reductions.

		size minimized			depth minimized				improv
		Frigate	**CBMC-GC**		**Prev. [10,30]**		**ShallowCC**		
Circuit	n	s^{nX}	s^{nX}	d^{nX}	s^{nX}	d^{nX}	s^{nX}	d^{nX}	d^{nX}
Building Blocks									
Add $n \rightarrow n$	32	31	31	31	232	11	159	**5**	**54 %**
Sub $n \rightarrow n$	32	31	61	31	232	11	159	**5**	**54 %**
Mul $n \rightarrow 2n$	32	2,082	4,600	67	2,218	19	2,520	**15**	**21 %**
Mul $n \rightarrow n$	64	4,035	4,782	67	-	-	4,350	**16**	-
Arithmetics									
Div	32	1,437	2,787	1,087	7,079	207	5,030	**192**	**7 %**
Matrix 5x5	32	128,252	127,225	42	-	-	128,225	**17**	-
FloatAdd	32	-	2,289	164	1,820	59	2,437	62	-5 %
FloatMul	32	-	3,499	134	3,016	47	3,833	54	-14 %
Distances									
Hamming-160	1	719	371	9	-	-	281	**7**	-
Hamming-1600	1	4,691	7,521	31	-	-	1,021	**12**	-
2D-Euclidian	16	-	826	47	1,171	29	1,343	**19**	**34 %**
2D-Euclidian	32	-	3,210	95	3,605	34	5,244	**23**	**32 %**
2D-Manhatten	16	-	187	31	296	19	275	**13**	**31 %**
2D-Manhatten	32	-	395	63	741	23	689	**16**	**30 %**
Privacy Preserving Protocols									
BioMatch-32	16	-	88,385	1,101	-	-	90,616	**55**	-
BioMatch-1024	16	-	2.9M	35,821	-	-	2.9M	**90**	-
Ob.Array-32	8	-	803	66	248	5	538	**3**	**40 %**
Ob.Array-1024	32	-	100,251	2,055	32,736	10	65,844	**4**	**60 %**

techniques. The CSN detection shows its strengths for arithmetic functionalities. For example, the 5x5 matrix multiplication shows a depth reduction of 60 %, when optimizations are enabled. This is because the computation of a single vector element can be grouped into one CSN. The detection of reductions is a very specific optimization, yet, when applicable, the depth saving can be significant. When computing the minima of 100 integers, a depth reduction of 92 % is visible. Note that in this test the circuit size itself is unchanged, as only the order of multiplexers is changed. Gate level minimization is the most important optimization technique for all functionalities, which do not use all bits available in every program variable. In

Table 6. Comparison of circuit dimensions when compiled by ShallowCC with different optimization techniques enabled or disabled.

Circuit	n	w/o optimization size s^{nX}	w/o optimization depth d^{nX}	w/ optimization size s^{nX}	w/ optimization depth d^{nX}	Improvement size s^{nX}	Improvement depth d^{nX}
Optimization: Carry-Save Networks CSNs							
Matrix 5x5	32	143,850	42	128,225	17	**11 %**	**60 %**
4D-EuclidianDst	16	2,993	40	2,459	20	**18 %**	**50 %**
Optimization: Reduction							
Minima-100	16	5,742	594	5,742	42	0 %	**92 %**
BioMatch-1024	16	2,9M	7,181	2,9M	90	0 %	**98 %**
Optimization: Gate level minimization							
Hamming-160	1	5,389	77	281	7	**95 %**	**88 %**
FloatAdd	32	10,054	194	2,431	74	**75 %**	**61 %**

these cases constant propagation applies, which leads to significant reductions in size and depth, as exemplary shown for the floating point addition and computation of the Hamming distance. In general, when applicable, the optimization methods significantly improve the compiled circuits of ShallowCC.

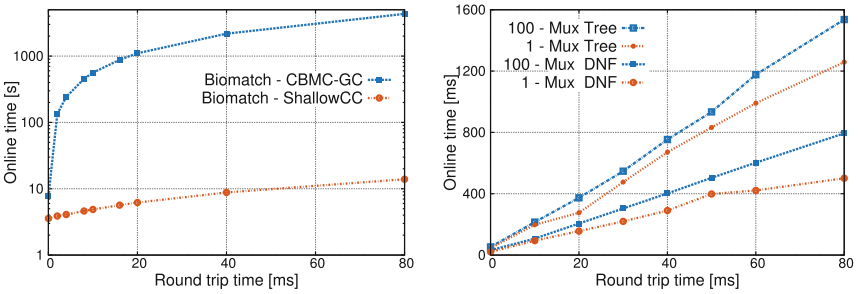
5.4 Protocol Runtime

To show that depth minimization improves the online time of MPC protocols, we evaluate a selection of circuits in the ABY framework [11]. ABY provides a state-of-the-art two-party implementation of the GMW protocol [14] secure in the semi-honest model. We extended the ABY framework by an adapter to parse ShallowCC's circuit format. For our experiments, we connected two machines, which are equipped with an AMD FX 8350 CPU and 16 GB of RAM, running Ubuntu 15.10 over a 1 Gbit ethernet connection in a LAN. To simulate different network environments we made used of the Linux network emulator *netem*.

In this experiment the *online* protocol runtimes of size and depth minimized circuits for different RTTs are compared. We omit timings of the pre-processing *setup* phase, as this pre-computation can take place independently of the evaluated circuits and with any degree of parallelism. We ran this experiment for different RTTs, starting with zero delay up to a simulated RTT of 80 ms.

The first functionality that we investigate is the biometric matching application with a database of 1024 entries. Here, we compare the circuits generated by CBMC-GC and ShallowCC. The resulting circuit dimensions are given in Table 5. The results, which are averaged over 10 runs, are given in Fig. 3a. We observe speed-ups of ShallowCC's circuit over CBMC-GC's circuit of a factor between 2 and 400, when increasing the RTT from ∼1 ms to 80 ms. A further comparison of size and depth optimized circuits is given in the full version of this paper.

The second functionality that we evaluate is the array read (MUX), which allows to analyze a size-depth trade-off. We compiled the read access to an array



(a) Biometric matching (b) 1024:1 multiplexer

Fig. 3. GMW protocol runtime of depth and size minimized functionalities. (a) compares the BioMatch functionality compiled with CBMC-GC and ShallowCC. (b) compares the depth-minimzed DNF and size-minimzed tree 1024:1 multiplexer, for a single and parallel execution. The resulting run times are plotted for different RTTs. We observe that the depth optimized circuits significantly outperform the size optimized circuits for any $RTT > 1$ ms.

with 1024× 32 bit integers. We compare the tree based MUX, as proposed in [30] with depth $d^{nX} = 10$ and size $s^{nX} = 32,736$ with our depth optimized MUX_{DNFd}, which has a depth of $d^{nX} = 4$ and size $s^{nX} = 65,844$ after gate level minimization. Each circuit is evaluated with ABY individually, as well as 100 times in parallel. This allows to also investigate whether single instruction multiple data (SIMD) parallelism, which is favored in GMW [11], has a significant influence on the results. The resulting online runtimes for both circuits are illustrated in Fig. 3b. All data points are averaged over 100 runs. We observe that for almost every network configuration beyond 1 ms RTT, the depth optimized circuits outperform their size optimized counterparts by a factor of two. The reason for the factor of two is, that the GMW protocol requires one communication round for the input sharing as well as one round for the output sharing, which leads to 6 communication rounds in total for the MUX_{DNFd} and 12 rounds for the tree MUX. Moreover, we observe that here applied data parallelism shows no significant effect on the speed-up gained through depth reduction.

In conclusion, the experiments support our introductory statement that depth minimization is of uttermost importance to gain further speed-ups in round-based MPC.

6 Conclusion

In this work we presented ShallowCC, the first depth-minimizing compiler that compiles a high-level language to Boolean circuits for MPC. We proposed and implemented multiple optimization techniques and presented newly optimized building blocks. ShallowCC is capable of compiling circuits that are up to 2.5 times shallower than hand optimized circuits and up to 400 times shallower than circuits compiled from size optimizing compilers, while still maintaining a competitive circuit size.

We note that ShallowCC is currently missing the support of an interpreted or mixed-mode language, which allows the efficient evaluation of very large applications. However, we are convinced that the combination of a mixed-mode interpreter, e.g., [21,24,26,35], with ShallowCC is mostly an engineering task rather than a research challenge and therefore leave it for future work.

Acknowledgments. This work has been co-funded by the DFG as part of project S5 within the CRC 1119 CROSSING, by the DFG as part of project A.1 within the RTG 2050 "Privacy and Trust for Mobile User", and by an Erwin Schrödinger Fellowship (Austrian Science Fund (FWF): J3696-N26).

References

1. Amini, M., Creusillet, B., Even, S., Keryell, R., Goubier, O., Guelton, S., McMahon, J.O., Pasquier, F.-X., Péan, G., Villalon, P.: Par4All: from convex array regions to heterogeneous computing. In: Workshop on Polyhedral Compilation Techniques (2012)
2. Ben-David, A., Nisan, N., Pinkas, B.: Fairplaymp: a system for secure multi-party computation. In: ACM CCS (2008)
3. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: ACM STOC (1988)
4. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: a framework for fast privacy-preserving computations. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 192–206. Springer, Heidelberg (2008)
5. Buchfuhrer, D., Umans, C.: The complexity of boolean formula minimization. J. Comput. System Sci. **77**, 1 (2011)
6. Büscher, N., Katzenbeisser, S.: Faster secure computation through automatic parallelization. In: USENIX Security (2015)
7. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
8. Damgård, I., Geisler, M., Krøigaard, M., Nielsen, J.B.: Asynchronous multiparty computation: theory and implementation. In: Jarecki, S., Tsudik, G. (eds.) PKC 2009. LNCS, vol. 5443, pp. 160–179. Springer, Heidelberg (2009)
9. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012)
10. Demmler, D., Dessouky, G., Koushanfar, F., Sadeghi, A., Schneider, T., Zeitouni, S.: Automated synthesis of optimized circuits for secure computation. In: ACM CCS (2015)
11. Demmler, D., Schneider, T., Zohner, M.: ABY - a framework for efficient mixed-protocol secure two-party computation. In: NDSS (2015)
12. Earle, J.: Latched carry-save adder. IBM Techn. Discl. Bull. **7**(10), 909–910 (1965)
13. Erkin, Z., Franz, M., Guajardo, J., Katzenbeisser, S., Lagendijk, I., Toft, T.: Privacy-preserving face recognition. In: Goldberg, I., Atallah, M.J. (eds.) PETS 2009. LNCS, vol. 5672, pp. 235–253. Springer, Heidelberg (2009)
14. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: ACM STOC (1987)
15. Harris, D.: A taxonomy of parallel prefix networks. In: IEEE ASILOMAR (2003)

16. Henecka, W., Kögl, S., Sadeghi, A., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: ACM CCS (2010)
17. Holzer, A., Franz, M., Katzenbeisser, S., Veith, H.: Secure two-party computations in ANSI C. In: ACM CCS (2012)
18. Keller, M., Scholl, P.: Efficient, oblivious data structures for MPC. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, Part II. LNCS, vol. 8874, pp. 506–525. Springer, Heidelberg (2014)
19. Kolesnikov, V., Sadeghi, A.-R., Schneider, T.: Improved garbled circuit building blocks and applications to auctions and computing minima. In: Garay, J.A., Miyaji, A., Otsuka, A. (eds.) CANS 2009. LNCS, vol. 5888, pp. 1–20. Springer, Heidelberg (2009)
20. Kolesnikov, V., Schneider, T.: Improved garbled circuit: free XOR gates and applications. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 486–498. Springer, Heidelberg (2008)
21. Kreuter, B., Shelat, A., Mood, B., Butler, K.: PCF: A portable circuit format for scalable two-party secure computation. In: USENIX Security (2013)
22. Kreuter, B., Shelat, A., Shen, C.: Billion-gate secure computation with malicious adversaries. In: USENIX Security (2012)
23. Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.W.: Automating efficient ram-model secure computation. In: IEEE S&P (2014)
24. Liu, C., Wang, X.S., Nayak, K., Huang, Y., Shi, E.: Oblivm: a programming framework for secure computation. In: IEEE S&P (2015)
25. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay - secure two-party computation system. In: USENIX Security (2004)
26. Mood, B., Gupta, D., Carter, H., Butler, K., Traynor, P.: Frigate: a validated, extensible, and efficient compiler and interpreter for secure computation. In: IEEE European Symposium on Security and Privacy (2016)
27. Mood, B., Letaw, L., Butler, K.: Memory-efficient garbled circuit generation for mobile devices. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 254–268. Springer, Heidelberg (2012)
28. Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 681–700. Springer, Heidelberg (2012)
29. Paterson, M.S., Pippenger, N., Zwick, U.: Optimal carry save networks
30. Schneider, T., Zohner, M.: GMW vs. Yao? efficient secure two-party computation with low depth circuits. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 275–292. Springer, Heidelberg (2013)
31. Songhori, E.M., Hussain, S.U., Sadeghi, A., Schneider, T., Koushanfar, F.: Tinygarble: Highly compressed and scalable sequential garbled circuits. In: IEEE S&P (2015)
32. Wallace, C.S.: A suggestion for a fast multiplier. IEEE Trans. Electron. Comput. **13**(1), 14–17 (1964)
33. Yao, AC.-C.: Protocols for secure computations (Extended Abstract). In: Annual Symposium on Foundations of Computer Science, FOCS 1982 (1982)
34. Zahur, S., Evans, D.: Circuit structures for improving efficiency of security and privacy tools. In: IEEE S&P (2013)
35. Zahur, S., Evans, D.: Obliv-c: a language for extensible data-oblivious computation. IACR Cryptology ePrint Archive 2015 (2015)
36. Zhang, Y., Steele, A., Blanton, M.: PICCO: a general-purpose compiler for private distributed computation. In: ACM CCS (2013)