Form Methods Syst Des (2017) 51:308–331
DOI 10.1007/s10703-017-0300-0

On compiling Boolean circuits optimized for secure multi-party computation

Niklas Büscher[1] · **Martin Franz**[1] · **Andreas Holzer**[2] · **Helmut Veith**[2] · **Stefan Katzenbeisser**[1]

Published online: 14 September 2017

Abstract Secure multi-party computation (MPC) allows two or more distrusting parties to jointly evaluate a function over private inputs. For a long time considered to be a purely theoretical concept, MPC transitioned into a practical and powerful tool to build privacy-enhancing technologies. However, the practicality of MPC is hindered by the difficulty to implement applications on top of the underlying cryptographic protocols. This is because the manual construction of efficient applications, which need to be represented as Boolean or arithmetic circuits, is a complex, error-prone, and time-consuming task. To facilitate the development of further privacy-enhancing technology, multiple compilers have been proposed that create circuits for MPC. Yet, almost all presented compilers only support domain specific languages or provide very limited optimization methods. In this work (this is an extended and revised version of the paper 'Secure Two-party Computations in ANSI C' (Holzer et al., in: ACM CCS, 2012) that reflects the progress in secure computation and describes the current optimization tool chain of CBMC-GC) we describe our compiler CBMC-GC that implements a complete tool chain from ANSI C to circuit. Moreover, we give a comprehensive overview of circuit minimization techniques, which we have identified and adapted for the creation of efficient circuits for MPC. With the help of these techniques, our compilation approach allows for a high level of abstraction from the cryptographic primitives used in MPC protocols, as well as the complex design of digital circuits. By using the model checker CBMC as a compiler frontend, we illustrate the link between MPC, formal methods, and digital logic design. Our experimental results illustrate the effectiveness of the implemented optimizations techniques for various example applications. In particular, compared with other state-of-the-art compilers, we show that CBMC-GC compiles circuits from the same source code that are up to four times smaller.

✉ Niklas Büscher
buescher@seceng.informatik.tu-darmstadt.de

✉ Stefan Katzenbeisser
katzenbeisser@seceng.informatik.tu-darmstadt.de

1 Technische Universität Darmstadt, Darmstadt, Germany

2 Technische Universität Wien, Vienna, Austria

Keywords Secure multi-party computation · Compiler · Logic synthesis

1 Introduction

In secure multi-party computation (MPC), two or more distrusting parties jointly evaluate a function over their inputs in such a way that each party keeps its input unknown to the other parties. MPC can be visualized by imagining a trusted third party (TTP), which receives the inputs of all parties, computes the desired functionality, and returns the result back to the input parties. Yet, instead of actually invoking a physical TTP, a cryptographic secure computation protocol is run that simulates such a TTP, while guaranteeing correctness and privacy. As such, MPC provides a generic approach for privacy-enhancing technologies by protecting sensitive data during computation steps on potentially untrusted platforms. For a long time seen only as a theoretical construction, many protocols and optimizations made MPC practical for various real world applications, e.g., electronic auctions [6] or privacy-preserving face recognition [18].

MPC protocols require an application (i.e., a functionality f) to be described in the form of Boolean (e.g., Yao's Garbled Circuits protocol [46], GMW [20]) or Arithmetic circuits (e.g., Sharemind [5], SPDZ [14]), quite similar to classic hardware design. During protocol runtime, each gate in the circuit is evaluated in software using different cryptographic instructions and communication patterns depending on the gate types. Hence, a circuit description can be seen as the assembly language of MPC protocols. In this work, we only focus on the creation of Boolean circuits.

1.1 Compilation for MPC

The manual construction of efficient Boolean circuits from a high-level functionality description is a complex, error-prone, and time-consuming task. Early MPC protocols (or the circuits used in early MPC protocols) were designed by hand. Clearly this approach is infeasible for any larger application. In this work, we describe our compiler CBMC-GC, the first compiler of ANSI C programs for MPC. CBMC-GC is a high-level synthesis tool that allows a programmer to write the function f to be computed by MPC in the form of a C program. Consequently, the programmer can view MPC as a dedicated hardware platform, and compile standard C programs, which are only required to have bounded loops to achieve obliviousness of the computation, to this platform. For this purpose, the CBMC-GC compiler transforms the C source code into a Boolean circuit optimized for MPC preserving the bit-precise semantics of C.

Technically, CBMC-GC is based on the software architecture of the model checker CBMC by Clarke et al. [11], which was designed to verify ANSI C source code. CBMC transforms an input C program f, including assertions that encode properties to be verified, into a Boolean formula B_f, which is then analyzed by a SAT solver. The formula B_f is constructed in such a way that the Boolean variables correspond to the memory bits manipulated by the program and to the assertions in the program. CBMC is an example of a *bit-precise* model checker, i.e., the formula B_f encodes the real life memory footprint of the analyzed program on a specific hardware platform under ANSI C semantics. (CBMC allows the user to configure the hardware platform, e.g., the word size.) The construction of the formula B_f moreover ensures that satisfying assignments found by the SAT solver are program traces that violate assertions in the program. Thus, CBMC is essentially a compiler that translates C source code into Boolean formulas. Our tool CBMC-GC (CBMC for Garbled Circuits)

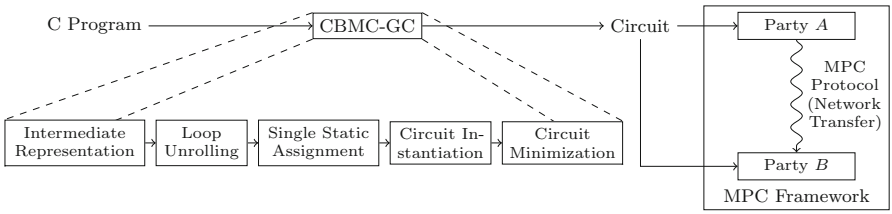

Fig. 1 Tool chain of CBMC-GC from C to circuits, ready for their use in secure computation, illustrated for the two party case. The compilation steps are discussed in Sect. 4.1

builds on the core engine of CBMC. We modify CBMC to transform a C program into a Boolean circuit rather than a formula. While CBMC optimizes the resulting formulas for easy solvability using a SAT solver, we changed the CBMC engine such that it outputs circuits suiting the requirements of Boolean circuit based MPC (described in Sect. 2.1). CBMC-GC's tool chain is illustrated in Fig. 1. The input source passes multiple compilation steps, such as parsing and transformation into an intermediate representation, discussed in detail in the next sections, before being converted into a Boolean circuit. This Boolean circuit can then be used in any MPC protocol (illustrated is a two-party secure computation protocol) as application description. Thus, compilation of circuits is separated from the actual MPC protocol and the circuit obtained by CBMC-GC is subsequently ready for use in any MPC framework.

1.2 Optimization

For almost all deployment scenarios of MPC protocols, the circuit has to be generated once, whereas the MPC protocol, evaluating the circuit, will be run multiple times. Hence, due to the high computational costs of MPC protocols compared to generic computation, it is very worthwhile to minimize the size of circuits used in MPC protocols. However, during compilation an efficient high-level description of a functionality does not necessarily translate to an efficient representation as a circuit. For example, in Sect. 5, we compare the circuit sizes when compiling a merge and a bubble sort algorithm. Merge sort has a commonly superior efficiency when computed on a RAM based architecture, whereas bubble sort compiles to a significantly smaller circuit representation. To provide another example, when writing source code for RAM based architectures, developers commonly rely on a few data types that are available, e.g., `unsigned int` or `long`. This leads to unused bits in a circuit, where arbitrarily bit-widths can be used. However, it is a tedious programming task to specify precise bit-widths for every variable to achieve minimal circuits.

To offer a level of abstraction from logic synthesis and to facilitate the use of MPC, compilers that allow the compilation of generic code into efficient circuits, henceforth called optimizing compilers, are of high interest. Thus, optimizing compilers should for example identify overly allocated bit-widths on the source code level and adjust them on the gate circuit level accordingly. Without advanced optimization techniques, the developer is required to be very familiar with logic synthesis for MPC and compiler internals to write code that compiles into an efficient circuit and thus, efficient application.

In this work, we also highlight optimization techniques implemented in CBMC-GC since its initial release that allow to compile efficient circuits for MPC from standard ANSI-C code. For the creation of optimized circuits, a fixed-point algorithm is employed that combines techniques from classic logic synthesis, such as constant propagation, SAT sweeping, and rewrite patterns to compile circuits that are significantly smaller than those generated by recent related work. For the exemplary computation of the Hamming distance we report circuit sizes

that are two times smaller, whereas for the compilation of a software implementation of IEEE compliant floating point addition, we report circuit sizes that are even four times smaller. The created circuits are comparable in size when adapting state-of-the-art commercial tool chain from a hardware description language (HDL) to circuits for MPC [43]. However, the authors of [43] also observed that when extending their tool chain towards high-level synthesis, which is comparable to the functionality of CBMC-GC, the resulting circuits are three to nine times larger than the circuits created from an HDL description. Consequently, our non-standard compilation approach represents a promising alternative for the creation of efficient circuits.As the complexity of most MPC protocols scales linearly with the circuit size, an improvement in circuit size directly translates into a similar improvement in computation and communication costs when evaluating the circuit in MPC.

1.3 Outline

In the remaining part of this paper, we first discuss how secure computation can be realized over Boolean circuits in Sect. 2 and illustrate the need for optimizing compilers. Then, we present a survey on existing compilers for MPC and compare these compilers regarding their optimization techniques in Sect. 3. We introduce CBMC-GC's compilation chain and describe its optimization techniques in Sect. 4. Finally, we present an evaluation for various example applications in Sect. 5 before concluding our work in Sect. 6.

2 Preliminaries

In this section we give an introduction into Boolean circuit based MPC, before illustrating the need for optimizing compilers that create circuits for MPC based on high-level language descriptions (high-level synthesis).

2.1 Secure computation over Boolean circuits

Yao's garbled circuits [45,46], Goldreich–Micali–Wigderson (GMW) [20] and many other secure computation protocols, such as [30,38], have in common that they operate on functionality descriptions in form of Boolean circuits. Similar to digital circuits, a Boolean circuit for MPC is a directed acyclic graph (DAG) consisting of Boolean gates (Boolean functions). For each gate in the circuit the protocols execute cryptographic instructions, depending on the particular gate type.

A circuit for MPC has two sets of input wires, representing the input of each party, and one or more sets of output wires, representing the output of the computation. In most protocols, the output wires can be shared between all parties, or used as individual outputs for each party. Commonly, circuits in MPC consist of two-input (binary) gates, e.g., AND, XOR, and (unary) inverters, whose output can be used as input to multiple subsequent gates. Circuits with binary gates allow a very generic use, as they can be evaluated in almost all MPC protocols.

2.2 Cost model

In MPC protocols different gates have different evaluation costs. Typically, the evaluation of non-linear gates (i.e., gates that cannot be represented by a Boolean function that computes a linear combination of its inputs, e.g., AND, NAND and OR) is costly in the number of cryptographic operations and communication effort, while the evaluation of linear gates (e.g., XOR) is essentially free [20,26]. This distinction between linear and non-linear gates holds

for most known MPC protocols, independent of the underlying cryptographic paradigm, such as circuit garbling, linear or Boolean secret sharing. Therefore, to achieve efficiency when creating circuits for MPC, the main optimization goal is to produce circuits with a minimal number of non-linear gates.

2.3 Notation

We notate the non-linear size of a circuit with s, i.e., the total number of non-linear (non-XOR) gates of a circuit. Furthermore, we denote bit strings in lower-case letters, e.g., x. We refer to individual bits in capital letters X and denote their negation with $\overline{X}$. We refer to a single bit at position i within a bit string as 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.

We remark, that some MPC protocols only provide protocols for AND and XOR gates. Yet, circuit descriptions with more extensive gate types can be emulated by a combination of AND, XOR, and the constant wire '1'. Moreover, with the practically free evaluation of linear gates, other gate types can be substituted with no additional evaluation costs when not available. For example, the unary inverter can be expressed as $\overline{X} = 1 \oplus X$ or the binary OR operation can be substituted by two XOR and one AND $A + B = (A \oplus B) \oplus (AB)$.

2.4 Two's complement

The two's complement is the common representation of signed binary numbers in hardware. This representation has the advantage that arithmetic operations for unsigned binary numbers such as addition, subtraction, and multiplication can be reused. In the two's complement, negative numbers are represented by flipping all bits and adding one: $-x = \overline{x}+1$. In the following sections, we assume a two's complement representation, when referring to negative numbers.

2.5 The need for optimization in compilation for MPC

To illustrate the need for optimization in circuit compilation from high-level code, we make use of a simple example code snippet given in Listing 1. The example begins in Line 7, where an input variable is declared that is only instantiated during protocol evaluation. Hence, its actual value is unknown during compile time. Next, a variable `t` is declared and initialized with a constant value of 43,210. Then, a helper function, which is declared in Line 1, is called that checks whether the given argument is an odd number. Depending on the result of the helper function, `t` will be incremented by one in Line 10.

Listing 1 Code example to illustrate the need for optimization on the gate level.

```
1  int is_odd(int val) {
2      return ((val & 1) == 1);
3  }
4
5  int main() {
6    [...]
7    int INPUT_A_x;
8    int t = 43210;
9    if(is_odd(INPUT_A_x) {
10     t = t + 1;
11   }
12   [...]
13 }
```

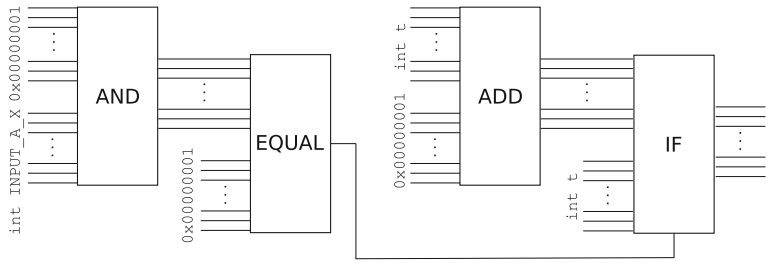

Fig. 2 Circuit after naïve translation from source code, consisting of four building blocks, namely a logical AND, an equivalence comparison (EQUAL), a binary adder (ADD), and a multiplexer (IF)

A naive translation of this code into a Boolean circuit would lead to a circuit consisting of four building blocks, namely, a logical AND operator, an integer equality check, an integer addition as well as a conditional integer assignment. Figure 2 illustrates the result of such a direct translation into a circuit. When using the best known building blocks (see Sect. 4.2) and assuming a standard integer bit-width of 32 bits, 31 or 32 non-linear gates are required for each building block. This results in a total circuit size of $s = 2 \cdot 31 + 2 \cdot 32 = 126$.

However, an optimizing compiler could detect that the comparison is only a single bit comparison, whose result is equal to the least significant bit of `INPUT_A_x`. Hence, no gate is required for the comparison. Moreover, because variable `t` is initialized by an even constant, the addition in Line 10 can be folded into an assignment of a constant, which leads to a circuit that consists only of a single wire and that does not even contain a single gate, namely:

```
t@0 = INPUT_A_x@0
```

there we use the notation `@`n to symbolize the accesses to the n-th bit, with `@0` being the least significant bit. We observe a significant difference in this simple example between naïve translation and optimized compilation. Such a size reduction directly relates to an improvement in the protocol's runtime, as the (amortized) computational cost of an evaluating MPC protocol scales linear with the circuit size.

To gain an optimized circuit from generic source code, compilers can follow two strategies. First, the compiler can use symbolic execution and abstract interpretation to compute constants as well as the actual used bit sizes. Second, the unoptimized circuit can be optimized on the gate level, by using various gate-level optimization techniques. Both approaches are applied by CBMC-GC.

2.6 Circuit minimization

Beside to the practical work on compilers for MPC, the isolated task of minimizing the non-linear complexity of Boolean circuits has also been studied in different areas of cryptography. In one of the first works on non-linear circuit complexity (also known as multiplicative complexity) Schnorr [41] showed by a counting argument that the circuit complexity of almost all Boolean functions on n variables is exponential in n. Interestingly, is has been shown by Turan and Peralta [44] that only $n-1$ non-linear gates are required for all functions with less than 6 variables. To the best of our knowledge, there is no known explicit function with super-linear circuit complexity. Furthermore, concrete circuits have been minimized for cryptanalysis or to achieve side-channel resistance in various works, e.g., [13,22].

3 Compilers for Boolean circuit based MPC

In this section, we survey existing compilers for Boolean circuit based MPC and illustrate the relevance in related field of cryptography. We observe that the compilation of Boolean circuits for MPC shares many similarities with classic hardware design. Yet, there are multiple subtle differences. First, both differ in their application. Circuits for MPC have a very generic use as interactive (end-user) applications, whereas classic hardware design mostly focuses on processing or accelerating units. Thus, for a widespread use of privacy-enhancing technology in form of MPC, high-level synthesis is desirable. Second, circuits for MPC protocols purely rely on combinational logic rather than sequential logic, because building blocks for reusable storage, such as Flip-flops, are not available.[1] Third, no physical properties of circuits have to be considered, e.g., such as the fan-out degree of gates, as the circuits in MPC are only evaluated virtually. Fourth, the costs for different types of gates differ significantly. In classic logic synthesis, Boolean NAND gates are favored over XOR gates due to their placement costs. However, as mentioned above, in most MPC protocols the evaluation of linear gates is practically for free, whereas the evaluation of non-linear gates is costly. Therefore, the main goal in circuit synthesis for MPC is to produce circuits with a minimal number of non-linear gates. Finally, even though not targeted in this work, we note that MPC is on the verge of handling circuits that are larger than those produced in classic hardware design. For example, a very recent accelerator of NVIDIA, the Tesla P100 Computing Platform, has 15 Billion (B) transistors.[2] Garbling a circuit with 15B logical gates with a semi-honest Yao's Garbled Circuit implementation (e.g., [2]) requires less than 20 min using a single core on a commodity CPU.

When introducing the first practical implementation of an MPC protocol, Malkhi et al [33] realized the need for tool support and presented a first compiler alongside their protocol implementation. Since then, multiple compiler prototypes for use in MPC have been proposed, as the size of realizable applications has grown. Since many of these compilers follow different approaches, in this section, we give a classification of (recently) published compilers that create circuits for Boolean circuit based MPC.

3.1 Compiler classification

Compilers for MPC can be categorized based on whether they compile from a (minimalistic) *domain-specific language (DSL)* or from a widely used *common programming language*. Moreover, compilers can be *independent* or *integrated* into an MPC framework. Integrated compilers produce an intermediate representation, which is *interpreted* (instantiated by a circuit) only during the execution of an MPC protocol. These interpreted circuit descriptions commonly allow a more compact circuit representation. Independent compilers create circuits independent from the executing framework, and thus have the advantage that produced circuits can be optimized to the full extent during compile time and are more versatile in their use in MPC frameworks. Some integrated compilers support the compilation of *mix-mode secure computation*. Mix-mode computation allows to write code that distinguishes between oblivious (private) and public computation. This leads to an even tighter coupling between compiler and execution framework, but allows to express a mix-mode program in a single language. Moreover, some integrated compilers support the compilation of programs for *hybrid*

[1] We note that MPC protocols can be combined with protocols for oblivious storage, e.g., ORAM [21], under various security and performance trade-offs. These constructions are beyond the scope of this work.

[2] https://devblogs.nvidia.com/parallelforall/inside-pascal/.

secure computation protocols, which combine different cryptographic approaches. The first example is the TASTY [23] compiler, which creates descriptions that combine homomorphic encryption and Yao's Garbled Circuits based computation. More recent compilers, e.g., ObliVM [32], combine Yao's Garbled Circuits with protocols for oblivious data structures.

We begin by giving an overview on compilers that use a DSL as input language. The Fairplay framework by Malkhi et al. [33] started research on practical MPC. Fairplay compiles a domain specific hardware description language called SFDL into a gate list for use in Yao's Garbled Circuits. Following Fairplay, Henecka et al. [23] presented the TASTY compiler, which compiles its own DSL, called TASTYL, into an interpreted hybrid protocol. The PAL compiler by Mood et al. [37] aims at low-memory devices as the compilation target. PAL also compiles Fairplay's hardware description language. The KSS compiler by Kreuter et al. [28] is the first compiler that shows scalability up to a billion gates and uses gate level optimization methods, such as constant propagation and dead-gate elimination. KSS compiles a DSL into a flat circuit format. TinyGarble by Songhori et al. [43] uses (commercial) hardware synthesis tools to compile circuits from hardware description languages such as Verilog or VHDL. On the one hand, this approach allows the use of a broad range of existing functionality in hardware synthesis, but also shows the least degree of abstraction, by requiring the developer to be experienced in hardware design. We remark, that high level synthesis from C is possible, yet, as the authors note, this leads to significantly less efficient circuits. Recently, Mood et al. [36] presented the Frigate compiler, which aims at very scalable and extensively tested compilation of another DSL. Frigate and TinyGarble produce compact circuit descriptions that compress sequential circuit structures.

Examples for mix-mode frameworks that compile from a DSL are L1, ObliVM, and Obliv-C. Similar to TASTY, the L1 compiler by Schröpfer et al. [42] compiles a DSL into a mixed-mode protocol including homomorphic encryption. ObliVM by Liu et al. [32] extends SCVM [31], which both compile a DSL that support the combination of oblivious data structures with MPC. This approach allows the efficient development of oblivious algorithms. Yet, both compilers provide only very limited gate and source code optimization methods. Obliv-C by Zahur and Evans [47] also supports oblivious data structures, but follows a different, yet elegant approach by compiling a modified variant of C into an executable application that also supports mix-mode computations.

The CBMC-GC compiler is the first of two compilers that creates circuits for MPC from a common programming language (ANSI-C). CBMC-GC follows the independent compilation approach and produces a single circuit description. Moreover, CBMC-GC applies source code optimization through symbolic execution and utilizes state of the art logic synthesis minimization methods. CBMC-GC has also been extended to compile depth-minimal circuits [8], required for multi-round MPC protocols, to compile parallel circuits for more efficient parallelization of MPC protocols [10], and also to optimize large scale circuits [9].

The PCF compiler by Kreuter et al. [27] also compiles C, using the portable LCC compiler as a frontend. PCF compiles an intermediate bytecode representation given in LCC into a interpreted circuit format. PCF shows greater scalability than CBMC-GC, yet only supports comparably limited optimization methods that are only applied locally for every function.

Mood et al. [36] provide a detailed experimental comparison of the aforementioned compilers and pointed out that reliability, correctness and gate-level optimizations are very limited for almost all compilers. In Table 1 we give a summary of the more recent compilers with a special focus on the source and gate level optimization methods. We distinguish their input language and their support for mix mode and interpreted languages. With the exception of CBMC-GC and TinyGarble, we observe that the compilers (including [36]) only provide limited gate level and source code optimization methods, in particular constant arguments

Table 1 Compiler comparison for Boolean circuit based MPC

Compiler	Language	Interpreted	Mix-mode	Maturity [36]	Const. prop./folding	Gate level opt.
CBMC-GC'12	ANSI-C	No	No	Yes[a]	Global	Dead gate elimination, constant propagation, theorem rewriting, SAT sweeping
KSS'12	DSL	No	No	No	No	Constant propagation, dead gate elimination
PCF'13	ANSI-C	Yes	No	No	Limited[b]	Local constant propagation, dead gate elimination
ObliVM'15	DSL	Yes	Yes	No	No	No
OblivC'15	DSL ANSI-C	Yes	Yes	No	Local	Local and limited [36] constant propagation, dead gate elimination
TinyGarble'15	Verilog VHDL	No[c]	No	Yes	n/a	Dead gate elimination, constant propagation, rewriting, SAT sweeping
Frigate'16	DSL	No[c]	No	Yes	Local	Local constant propagation, dead gate elimination

Compared is the source *language*, whether they compile to complete circuit or an intermediate representation that is *interpreted* during runtime, whether they support *mix-mode* MPC, their *maturity* (compilation correctness) based on the study by Mood et al. [36], and the applied source and gate-level *optimization* techniques

[a] Mood et al. [36] identified compilation aborts with the input/output notation that have been fixed in the current version of CBMC-GC

[b] PCF uses a frontend compiler LCC that generates optimized byte code for RAM based architectures

[c] TinyGarble and Frigate stores sequential building blocks in a compact way

in function calls are not propagated, which is essential when providing an programming interface to non MPC and digital logic experts.

4 From ANSI C to size minimized circuits

In this section, we first describe the architecture of our compiler CBMC-GC[3] (Sect. 4.1) that translates ANSI-C into circuits satisfying the requirements described above by adapting the software model checker CBMC [11]. Then, we describe the optimization techniques that have been identified as effective and that have been implemented in CBMC-GC for the creation of size minimized circuits for MPC, namely, efficient building blocks (Sect. 4.2) and gate-level minimization techniques (Sect. 4.3).

4.1 CBMC-GC's architecture

CBMC-GC uses the core engine of CBMC. Yet, instead of optimizing a Boolean formula towards its use in SAT solvers, it is optimized towards its use in MPC frameworks. Using a well tested bit-precise model checker equips CBMC-GC with a reliable compilation architecture. The compilation pipeline of CBMC-GC, illustrated in Fig. 1, is divided into five steps, where the first three steps are part of the standard CBMC processing, and the last two were adapted or have been added to optimize circuits for MPC. We begin with a description of the modifications of the input source code.

4.1.1 Code for CBMC-GC and circuit mapping

When programming for CPU/RAM architectures, inputs and outputs of a program are commonly realized with standard libraries that themselves invoke system calls of the operating systems. This is in contrast to MPC, where the input and output interface is defined by *input/output (I/O)* wires of the circuit. To realize the I/O mapping between C code and circuits, we use a special naming convention of I/O variables. The input variables are then left uninitialized in the source code and are only assigned a value during the evaluation of the circuit in an MPC framework. Hence, instead of adding additional standard libraries, CBMC-GC requires the developer to name input and output variables accordingly.

To illustrate this naming convention, we give an example source code of the millionaires' problem in Listing 2. The shown function is a standard C function, where only the input and output variables are specifically annotated as designated input of party P_A or P_B (Lines 2 and 3) or as output (Line 4). Hence, variables that are inputs of party P_A or P_B have to be named with a preceding `INPUT_A` or `INPUT_B`. Similar, output variable names have to start with `OUTPUT`. Aside from this naming convention, arbitrary C computations are allowed to produce the desired result, in this case a simple comparison (Line 5).

Listing 2 CBMC-GC code example for Yao's Millionaires' problem.

```
void millionaires() {
  int INPUT_A_income;          // Input Party A
  int INPUT_B_income;          // Input Party B
  int OUTPUT_result = 0;       // Output

  if (INPUT_A_income > INPUT_B_income) {
    OUTPUT_result = 1;
  }
}
```

[3] CBMC-GC is available at www.seceng.de/research/software/cbmc-gc/.

For simplicity, CBMC-GC only distinguishes between two parties and uses a shared output, which is the simplest case of secure two-party computation. However, this does not prevent the compilation of source code for more than two parties or code with outputs that are designated for specific parties only. This is because, during compilation CBMC-GC only distinguishes input and output variables, but not the association with any party. Hence, to compile code for more parties, an application developer can use its own naming scheme that extends the existing one.

As CBMC-GC outputs a mapping between every I/O variable and their associated wires in the circuit, this information can be used in every MPC framework to correctly map wires back to the designated parties.

Moreover, we remark that code for CBMC-GC must terminate in a finite number of steps. Therefore, CBMC-GC expects a number k as input which bounds the size of program traces (and CBMC-GC also determines if this bound is sufficient). This constraint is inherited from CBMC. However, for MPC this property is actually a mandatory requirement rather than a limitation. This is because combinatorial circuits have a fixed size and thus deterministic evaluation time in any MPC protocol. This is a logical consequence of the requirement that the runtime of a MPC protocol should not leak any information about the inputs of either party.

4.1.2 Compile chain

Given a source code as described above, it passes the following compilation steps of CBMC-GC:

1. *Intermediate representation* On input of an ANSI C program f and a bound k, CBMC(-GC) first translates the program into an intermediate representation—a so-called GOTO program. In a GOTO program, all control statements like while-loops are transformed into if-then-else statements (guarded GOTOs) with conditional jumps, similarly to assembly language.
2. *Loop unrolling* To make the program acyclic, the loops are replaced by a sequence of k nested `if` statements; the sequence is followed by a special assertion (called *unwinding assertion*) which can detect a missed loop iteration due to insufficient k. Similarly, recursive function calls are expanded k times. This process is called "unwinding" the program. For programs with at most k steps, unwinding preserves the semantics of the program.
3. *Single static assignment* Once the program is acyclic, it is turned into single-static assignment (SSA) form. This means that each variable x in the program is replaced by fresh variables $x_1, x_2, \ldots$ where each of them is assigned a value only once. For instance, the code sequence

   ```
   x=x+1; x=x*2;
   ```

   is replaced by

   $$x_2 = x_1 + 1; \quad x_3 = x_2 * 2;$$

   The SSA format has the important advantage that we can now view the assignments to program variables as mathematical equations. (Note that, an equation, such *x=x+1;*, is unsolvable.) The indices of the variables essentially correspond to different intermediate states in the computation.

4. *Circuit instantiation* In the next step, CBMC replaces the variables by bit vectors. For instance, depending on the architecture, an integer variable will be represented by a bit vector of size 16 or 32. For more complex variables such as arrays and pointers, CBMC uses more advanced techniques [12] whose presentation we omit for simplicity.
   Correspondingly, operations over variables (e.g. arithmetic computations or comparisons) are naturally translated into Boolean functions over the corresponding variables. Internally, CBMC realizes these Boolean functions as circuits, henceforth called *building blocks*, whose construction principles are inspired by methods from hardware design.
   At some places in this compilation step, we had to modify the circuit generation of CBMC for CBMC-GC for a subtle reason: Since CBMC aims to produce good instances for a SAT solver, it has the freedom to use circuits which are equisatisfiable with the circuits we expect, but not logically equivalent. CBMC sometimes introduces circuits with free input variables, and adds constraints which requires them to coincide with other variables. In these places, we had to change the circuit generation to reflect actual computation.
5. *Gate-level minimization* In the last step of the compilation chain, the instantiated circuit is minimized on the gate-level using a fix-point minimization procedure introduced by CBMC-GC.

More details on the first three compilation steps can be found in [12], whereas the building blocks optimized for MPC, as well as the minimization procedure used in CBMC-GC are elaborated upon in the following sub-sections.

4.2 Building blocks for Boolean circuit based MPC

Optimized building blocks are an essential part of 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 give a comprehensive overview over the currently best known building blocks for MPC based on Boolean circuits that are required for basic arithmetic and control flow operations.

4.2.1 Adder

All 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 best known constructions for computing the sum is by XOR-ing all inputs $S = A \oplus B \oplus C_{in}$, the carry-out bit can be computed by $C_{out} = (A \oplus C_{in})(B \oplus C_{in}) \oplus C_{in}$ [26]. Both, the HA and FA have size $s = 1$.

An n-bit *adder* takes two bit strings x and y of length n, representing two (signed) integers, as input and returns their sum as an output bit string S of length $n+1$. We note that according to the semantics in ANSI-C an addition is computed as $x + y \mod 2^n$ and no overflow bit is returned. The standard and best known adder is the Ripple Carry Adder (RCA) that consists of a successive composition of n FAs. This leads to a linear circuit size $s_{RCA} = n$ [25].

4.2.2 Subtractor

A subtractor for two n bit strings can be implemented with one additional non-linear gate by using the two's complement representation $x - y = x + \overline{y} + 1$, with $\overline{y}$ being the negated binary representation. The addition of negative numbers in the two's complement is equivalent to an addition of positive numbers.

4.2.3 Comparator

An *equivalence (EQ)* comparator checks whether two input bit strings of length n are equivalent and outputs a single result bit. The comparator can be implemented naïvely by a successive OR composition over pairwise XOR gates that compare single bits. This results in a size of $s_{EQ}(n) = n-1$ gates [26]. A *greater-than (GT)* comparator that compares two integers can be implemented with the help of a subtractor by observing that $x > y \Leftrightarrow x - y - 1 \geq 0$ and returning the carry out bit, which yields to a circuit size of $s_{GQ}(n) = n$.

4.2.4 Multiplier

In classic hardware synthesis, a multiplier (MUL) computes the product of two n bit strings x and y, which has a bit-width of $2n$. However, in many programming languages, e.g., ANSI-C, multiplication of unsigned numbers is defined as an $n \rightarrow n$ bit operation: $x \cdot y \mod 2^n$. The standard approach for computing an $n \rightarrow 2n$ bit multiplication is often referred to as the "school method". Using a bitwise multiplication and shifted addition, the product is computed as $\sum_{i=0}^{n-1} 2^i (X_i\, y)$. This approach leads to a circuit requiring n^2 1-bit multiplications and $n-1$ shifted n-bit additions, which in total results in a circuit size of $s_{MUL} = 2 \cdot n^2 - n$ gates [25]. When compiling a $n \rightarrow n$ bit multiplication with the same method, only half of the one bit multiplications are relevant, leading to a circuit size of $s_{MUL} = n^2 - n$ gates. The $n \rightarrow n$ multiplication of negative numbers in the two's complement representation can be realized with the same circuit.

Alternatively, for $n \rightarrow 2n$ bit multiplication, the Karatsuba-Ofmann multiplication (KMUL) can be used, achieving an asymptotic complexity of $\mathcal{O}(n^{log_2(3)})$. Henecka et al. [23] et al. presented the first adoption for MPC, which was subsequently improved by Demmler et al. [16] by 3% using commercial hardware synthesis tools. Their construction outperforms the school methods for bit-widths $n \geq 19$.

4.2.5 Multiplexer

Control flow operations, e.g., branches and array read access, are expressed on the circuit level through multiplexers (MUX). 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. 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. For example, a dynamic access to an array with four elements can be realized as illustrated in Fig. 3, where an array `a` is accessed with index `i`.

Kolesnikov and Schneider [26] 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 bit as $O =$

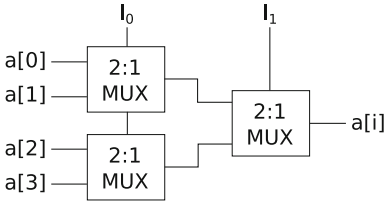

Fig. 3 Exemplary array access compiled into a multiplexer tree

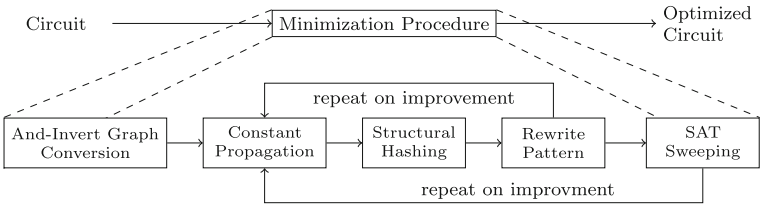

Fig. 4 Illustration of CBMC-GC's minimization procedure and its components

$(D^0 \oplus D^1)C \oplus D^0$. This leads to a circuit size for an n-bit 2:1 MUX of $s_{MUX}(n) = n$. The circuit size of a tree based m:1 MUX depends on the number of choices m, as well as the bit-width n: $s_{MUX\_tree}(m, n) = (m - 1) \cdot s_{MUX}(n)$.

4.2.6 Divisor

A divisor computes the quotient and/or remainder for a division of two binary integer numbers. The standard approach for integer division is known as long division and works similar to the school-method for multiplication. Namely, the divisor is iteratively shifted and subtracted from the remainder, which is initially set to the dividend. Only if the divisor fits into the remainder, which is efficiently decidable by overflow free subtraction, a bit in the quotient is set and the newly computed remainder is used. Thus, a divisor can be built with the help of n subtractors and n multiplexers, each of bit-width n, leading to a circuit size of $s_{SDIV}(n) = 2n^2$. The divisor can be improved by using restoring division [39], which leads to a circuit size of $s_{RDIV}(n) = n^2 + 2n + 1$.

4.3 Circuit minimization in CBMC-GC

As identified in Sect. 1, with the compilation being a one-time task, it is very useful to invest computing time in circuit optimization during their compilation. Moreover, as shown in Sect. 2.5, a naïve translation of code into optimized building blocks does not directly lead to a minimal circuit. Therefore, after the instantiation of all building blocks, and thus, construction of the complete circuit, a circuit minimization procedure is run that reduces the number of non-linear gates. Unfortunately, finding a minimal circuit for a given circuit is known to be Σ_2^P complete [7]. Because of this, CBMC-GC follows an heuristic approach using a minimization procedure that applies multiple different algorithms, known from logic synthesis, to reduce the non-linear circuit size. In the next paragraphs, we first describe the general procedure, before discussing the different components in detail.

4.3.1 CBMC-GC's minimization procedure

CBMC-GC's minimization routine is illustrated in Fig. 4. It begins with the translation of the circuit into an intermediate And-Invert Graph (AIG) representation, which is a circuit description that has been shown to allow gate-level optimizations in a very efficient manner. During the AIG translation, a first optimization is performed, before the main fix-point minimization routine is started. The algorithm is run until no further improvements in the circuit size are observed or a user given time bound has been reached. In both cases, the result of the latest iteration is returned.

In every iteration of the algorithm a complete and topological pass over all gates from from inputs to outputs is initiated. During this pass, constants, i.e., zero or one, are propagated (*constant propagation*), duplicated gate structures are eliminated (*structual hashing*) and small sub-circuits are matched and replaced by hand-optimized sub-circuits (*rewrite patterns*). If any improvement, i.e., reduction in the number of non-linear gates, is observed, a new pass is initiated. If no improvement is observed, a more expensive optimization routine is invoked that detects constant and duplicated gates using a SAT solver (*SAT sweeping*).

4.3.2 AIG, constant propagation and structural hashing

An AND-inverter graph is a representation of a logical functionality using only binary AND gates (nodes) with (inverted) inputs. AIGs have been identified as a very useful representation for circuit minimization, as they allow very efficient graph manipulations, such as the adding or merging of nodes. This phase is implemented with the ABC library [1], which provides state-of-the-art logic synthesis methods. As a first step in CBMC-GC, input and output wires (known as primary inputs and primary outputs) are created for every input and output variable. Then, during the instantiation of building blocks, the AIG is iteratively constructed by substituting every gate type that is different from an AND gate by a Boolean equivalent AIG subgraph. For example an XOR gate ($A \oplus B$) can be replaced by the following AIG: $\overline{\overline{A} \cdot \overline{B}} \cdot \overline{A \cdot B}$.

Whenever a node is added to the AIG, two optimization techniques are directly applied by the ABC library. First, constant inputs are propagated. Hence, whenever an input to a new node is known as constant, the added node is replaced by an edge. Second, structural hashing is applied. Structural hashing [15] is used to detect redundant sub-graphs that operate on the same inputs. The duplication check can be realized efficiently by hash-based comparison of the inputs during the insertion of new gates.

4.3.3 Rewrite patterns

Circuit rewriting is a greedy optimization algorithm used in logic synthesis [35], which was first proposed for hardware verification [4]. A rewrite pattern consist of two templates, i.e., sub-circuits, that are functionally equivalent, where the first is the template that is searched for and the second is the substitute. Pattern based rewriting has been shown to a very effective optimization technique in logic synthesis, as it can be applied with very little computational cost [35]. In CBMC-GC's compilation chain, rewrite patterns are of high importance due to multiple reasons. First, they are responsible for translating the AIG back into a Boolean circuit representation with a low number of non-linear gates. This is necessary, because all linear gates have been replaced by non-linear AND gates during the translation in the AIG representation. Second, pattern based rewriting allows for further MPC specific optimizations by applying patterns that favor linear gates and reduce the number of non-linear gates beyond a conversion from AIG. Finally, in CBMC-GC each rewrite pass is also used for constant propagation and structural hashing, as described above. To illustrate the importance of both techniques, we remark, that CBMC-GC performs a high-level synthesis, which, for example, frequently requires to reduce the bit-width of many declared variables through constant propagation.

For circuit rewriting in CBMC-GC, all gates are first ordered in topological order by their circuit depth. Subsequently, by iterating over all gates, the patterns are matched against all gates and possible sub-circuits. Whenever a match is found, the sub-circuit becomes a candidate for a replacement. However, the sub-circuit will only be replaced, if the substitution

Table 2 Example rewrite patterns that are used in CBMC-GC, grouped in three categories

Search pattern	Substitute
Propagate pattern	
$\overline{0}$	1
$0 \cdot A$ or $A \cdot 0$	0
$0 + A$ or $A + 0$	A
$0 \oplus A$ or $A \oplus 0$	A
$\overline{1}$	0
$1 \cdot A$ or $A \cdot 1$	A
$1 + A$ or $A + 1$	1
Trivial patterns	
$A \cdot A$	A
$A \cdot \overline{A}$	0
$A + A$	A
$A + \overline{A}$	1
$A \oplus A$	0
$A \oplus \overline{A}$	1
$\overline{\overline{A}}$	A
XOR patterns	
$\overline{A} \oplus \overline{B}$	$A \oplus B$
$(A + B) \oplus (A \cdot B)$	$A \oplus B$
$(A + B) \cdot \overline{(A \cdot B)}$	$A \oplus B$
$\overline{\overline{A} \cdot \overline{B}} \cdot \overline{A \cdot B}$	$A \oplus B$
$(A \cdot (B \oplus (A \cdot C)))$	$A \cdot (B \oplus C)$
$(A \cdot B) \oplus (A \cdot C)$	$(A \oplus C) \cdot A$
$\overline{(A \cdot C) \oplus (B \cdot C)}$	$\overline{(A \oplus B)} \cdot C$

leads to an actual improvement in the circuit size. This is guaranteed for the patterns themselves, which are designed to be minimizing. However, dependencies of intermediate gates, which might be input to other gates, can rule the substitution ineffective. In these cases the sub-circuit will not be replaced.

The outcome and performance of this greedy replacement approach depends not only on the patterns themselves, but also on the order of patterns. Therefore, in CBMC-GC, small patterns are matched first, e.g., single gate patterns, as they can be matched with little cost and offer guaranteed improvements, before matching more complex patterns that require to compare sub-circuits consisting of multiple gates and inputs. Table 2 lists exemplary rewrite patterns that are used in CBMC-GC and that have been shown to be very effective in our evaluation. In total more than 80 patterns are used for rewriting.

4.3.4 SAT sweeping

SAT sweeping is a powerful minimization tool, widely used for equivalence checking of combinatorial circuits [29,34]. The core idea of SAT sweeping is to prove that the output of a sub-circuit is either constant or equivalent to another sub-circuit (detection of duplicity). In both cases the sub-circuit is unnecessary and can be removed. As usual, SAT sweeping is applied in CBMC-GC in a probabilistic manner. A naïve application, which compares every

possible combination of sub-circuits would result in infeasible computational costs. Thus, for efficient equivalence checking, the circuit is first evaluated (simulated) multiple times with different (random) inputs. The outputs of all gates in every run are are then grouped by their output. Gates that always output one or always output zero are presumed constant, whereas gates that have the same output are presumed equivalent. This is then proven using the efficient tool of a SAT solver. For this purpose, sub-circuits have to be converted into conjunctive normal form. Due to its high computational costs in comparison with the circuit rewriting, SAT sweeping is only applied if other optimization methods cannot minimize the circuit any further.

5 Evaluation

In this section, we evaluate CBMC-GC's compilation approach alongside various example applications that emerged as standard benchmarks for MPC. We first present a comparison to previous releases of CBMC-GC (Sect. 5.2) to illustrate the effectiveness of the described optimization techniques. Moreover, we also present a comparison of the circuits generated by CBMC-GC with circuits from other recently presented compilers (Sect. 5.3).

We begin with a description of sample applications used for benchmarking purposes.

5.1 Benchmarking applications

All the following example applications have emerged as benchmarks for MPC frameworks or compilers.

5.1.1 Distances

Various distances need to be computed in many 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., [24,28,36]. The Hamming distance can be parametrized by the bit length of the input strings. We compare an implementation optimized for CPU/register based computation, a naïve implementation that compares and aggregates individual bits, and a tree based aggregation. The source code of the three implementations is given in "Appendix A". The *Manhattan* distance $dist_{MH} = |x_1 - x_2| + |y_1 - y_2|$ between two 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 *Euclidean* distance is defined as $dist_{ED} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. For the computation of the square root function, we implemented the Babylonian method (division by mean) over 5 and 20 iterations.

5.1.2 Biometric matching (BioMatch)

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 [18]. One of the main concepts is the computation of a distance, e.g., the Euclidean distance, between the sample and all database entries. Once all distances have been computed, the minimal distance determines the best match. In the following experiments, we fix the dimension of a sample to $d = 4$, as it has been used before [10,17]. We use a database of 128 and 256 samples consisting of four 32-bit integers. Due to the complexity of the square root function, it is common in MPC to use the squared Euclidean distance for matching [40].

The BioMatch application is very interesting for benchmarking, as it involves many parallel arithmetic operations, as well as a large number of comparisons.

5.1.3 Fixed and floating point operations

Fixed point and floating point arithmetic are necessary for all applications where numerical precision is required, e.g., privacy preserving statistics. We rely on the compiler's capabilities to compile a software fixed point and a IEEE-754 (32 bit) compliant software floating point addition and multiplication. Both implementations use the integer data types available in C to "emulate" the computation of fixed or floating point operations. Floating point operations are very suited to benchmark the gate level optimization methods of compilers since they require many bit operations.

5.1.4 Location-aware scheduling

Privacy-preserving availability scheduling is another use-case for MPC, e.g., [3]. The functionality matches the availability of two parties over a number of time slots, without revealing the individual schedule to the other party. Location-aware scheduling, as implemented here, also considers the location and maximum travel distance of the two parties for a given time slot. Therefore, the functionality outputs a matching time slot where both parties can meet and the distance between the parties is minimal. We benchmark the functionality for 56 slots with 16 bit integer coordinates, and distinguish between the squared Euclidean (*SQ*), as well as the Manhattan distance (*MH*).

5.1.5 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 [10,16, 24,27]. We evaluate a 5×5 and 8×8 matrix multiplication over 32 bit signed integers, fixed-point and floating point numbers.

5.1.6 Median computation

The secure computation of the median is required in privacy preserving statistics. It is an interesting task for compilation, as it involves the implementation of a sorting algorithm, as a sorted array allows easy access to the median element. Here we use two different sorting algorithms, namely bubble sort and merge sort. In contrast to generic computation, bubble sort compiles to a smaller circuit representation than merge sort. Here we use an array of $n = 21$ and $n = 31$ integer elements.

5.1.7 Random arithmetic operations

Circuits for programs that perform several arithmetic operations sequentially scale linear in the number of operations. We use programs, as in [24], that contain 3000 random arithmetic operations, comprising of 90% additions and 10% multiplications with varying numbers of input and output variables.

Table 3 Experimental results: circuit sizes in the number of non-linear gates produced by CBMC-GC v0.8 [24], v0.9 [19], and the current version for various example applications

Application	#gates v0.8	#gates v0.9	#gates current
Arithmetic operations 2000	405,640	319,584	253,776
Hamming distance (reg), 320 bit	6038	924	924
Hamming distance (reg), 800 bit	15,143	2344	2340
Hamming distance (reg), 1600 bit	30,318	4738	4726
Matrix multiplication, 5×5	221,625	148,650	127,255
Matrix multiplication, 8×8	907,776	600,768	522,304
Median, merge sort, 21 elements	244,720	136,154	61,403
Median, merge sort, 31 elements	602,576	348,761	152,823
Median, bubble sort, 21 elements	112,800	40,320	10,560
Median, bubble sort, 31 elements	349,600	89,280	23,040
Scheduling, sq. Euclidean, 56 items	317,544	169,427	113,064
Scheduling, Manhattan, 56 items	133,192	88,843	62,188

5.2 Evaluation of circuit minimization techniques

We abstain from an individual evaluation of the optimization methods presented in this work. The main reason is that the (de-)activation of a single optimization method has various side-effects that are hard to be controlled and measured in an isolated manner. For example, when using less efficient building blocks, the circuit minimization phase will partly be able to compensate inefficient building blocks and start optimizing these. However, the computation time spent on optimizing the building blocks can consequently not be applied to the remaining parts of the circuit. Similarly, SAT sweeping is highly ineffective without efficient constant propagation. Moreover, some rewrite patterns can become ineffective without the application of other rewrite patterns. Yet, we show that the combination of optimization techniques in CBMC-GC has continuously evolved, such that the circuit sizes compared to earlier releases have significantly been reduced.

Table 3 gives a comparison of circuit sizes between the first release of CBMC-GC v0.8 [24] in 2012, its successor CBMC-GC v0.9 [19] from 2014 and the current version described in this article. For comparison, we use the applications introduced in Sect. 5.1. All applications have been compiled from the same source code and optimized with a maximum optimization time of at most 10 min on a commodity laptop. The initial release of CBMC-GC provided no gate-level minimization techniques, yet it contained first optimized building blocks. In CBMC-GC v0.9 the optimization algorithm, described in this chapter, was introduced. All techniques and building blocks have subsequently been improved and adapted for the current version. For example the building blocks have been revisited and the set of rewrite patterns has been refined, which results in size reductions for all applications.

We observe that the improvement in building blocks is directly visible in the example application of random arithmetic operations and matrix multiplication, which have been improved up to a factor of two, between the first and the current release of CBMC-GC. These two applications purely consist of arithmetic operations that utilize the full bit-width of the used data types, and thus, barely profit from gate-level optimizations. The resulting circuit sizes for the Hamming distance computation show significant improvements when comparing the first and the current release, yet only marginal improvement in comparison to CBMC-GC v0.9. The more complex applications, such as bubble sort based median computation, which involve more interesting control flow logic, have been improved by more than a factor of ten between the first and the current release of CBMC-GC. Similarly, for the location

Table 4 Experimental results: compiler comparison between Frigate [36], OblivC [47] and the current version of CBMC-GC. Given are the circuit sizes in the number of non-linear gates when compiling various applications

Application	Frigate [36]	OblivC [47]	CBMC-GC	Improv. (%)
Biometric matching 128	561,218	560,192	404,419	**27.8**
Biometric matching 256	1.1 M	1.1 M	831,846	**25.7**
Euclidean 5, int	7960	7811	6235	**20.2**
Euclidean 20, int	25,675	25,001	23,255	7.0
Euclidean 5, fix	26,430	26,307	7834	**70.2**
Euclidean 20, fix	90,735	90,057	24,559	**12.3**
Float addition	5237	5581	1201	**77.1**
Float multiplication	16,502	14,041	3534	**74.8**
Hamming 160 (reg)	567	899	449	**20.8**
Hamming 1600 (reg)	6546	9269	4738	**27.6**
Hamming 160 (tree)	747	4929	351	**53.0**
Hamming 1600 (tree)	8261	49,569	3859	**53.3**
Hamming 160 (naïve)	1009	4929	541	**46.3**
Hamming 1600 (naïve)	10,282	49,569	6042	**41.2**
Matrix mult. 5×5, int	127,477	127,225	127,225	0
Matrix mult. 5×5, fix	314,000	313,225	183,100	**41.5**
Matrix mult. 5×5, float	2.7 M	2.4 M	626,506	**73.9**
Scheduling SQ 56	133,216	181,071	113,064	**15.1**
Scheduling MH 56	79,008	77,023	58,800	**23.7**

Marked in bold are significant improvements over the best previous result

aware scheduling application, which involves minimizable arithmetic operations as well as control flow logic, we observe an improvement up to a factor of three between the first and the current release.

5.3 Compiler comparison

We compare the circuits created by CBMC-GC for the aforementioned benchmark applications with the circuits created by the recently published Frigate [36] and Obliv-C [47] compilers, which are the most promising candidates for a comparison, as they create circuits with the least number of non-linear gates according the compiler analysis by Mood et al. [36]. TinyGarble [43] with its state-of-the art synthesis tool chain would be very interesting candidate for comparison, but unfortunately its complete C-to-circuit tool chain has not been open sourced. Therefore, here we only compare circuits created Frigate and Obliv-C with those created by CBMC-GC. Even though all compilers use different input languages, we ensure a fair comparison by implementing the functionalities using the same code structure (i.e., functions, loops), data types and bit-widths. All applications have been compiled with the latest available version at the time of writing. For the CBMC-GC, we again set a maximum optimization time limit of 10 min on a commodity laptop. The resulting circuit sizes and the improvement of CBMC-GC over the best result from related work are presented in Table 4. Circuit sizes above one million (M) gates are rounded to the nearest 100,000.

We observe that the current version of CBMC-GC outperforms related compilers in circuit size for almost all applications. For example, the biometric matching application, or schedul-

ing applications improve by more than 25%. Most significant is the advantage in compiling floating point operations, where a 77% improvement can be observed for the dedicated multiplication operation. A similar improvement is achieved for the computation of the Euclidean distance or matrix multiplication on non-integer values. Floating point and fix point operations are dominated by bit wise operations and thus, can be optimized with gate-level optimization when compiled from high-level source code. No improvement is observed for the integer based matrix multiplication. As discussed above, the matrix multiplication compiles into a sequential composition of building blocks, which barely can be improved further.

The Hamming distance computations show implementation dependent results and illustrate the challenges of circuit compilation from standard code. The source code of the three benchmarked implementations is given in "Appendix A". The tree based computation shows the smallest circuit size, even though it is the most inefficient CPU implementation. This is because, a tree-based composition allows to apply adders with small bit-widths for a majority of the bit counting. Comparing the resulting circuit sizes of the compilers, we observe differences between the implementations. The tree based and naïve implementation are significantly more optimized by CBMC-GC, i.e., up to a factor of two, than in the other compilers. The implementation optimized for register based computation compiles into a circuit that is also smaller in CBMC-GC than in related work, yet only by 20%. This is because, the compilation of the naïve bit counting profits significantly from constant propagation, as only a few bits per expression are required on the gate-level. The register optimized implementation maximizes the number of bits used per arithmetic operation, thus, allows only little gate-level optimization. We remark that Frigate and ObliviC compile larger applications noticeably faster than CBMC-GC, yet the circuits created by CBMC-GC are up to a factor of four smaller.

6 Conclusion

Secure computation is a very powerful cryptographic tool that allows to achieve privacy in a provable manner for almost all interactive applications. However, its wide-spread use crucially depends on its accessibility for non-domain experts. Optimizing compilers that compile high-level source code into efficient circuit descriptions, understood by MPC protocols, offer the required level of abstraction to facilitate the use of MPC. In this work, we have shown that CBMC-GC, which is based on the model checker CBMC, is well suited to compile circuits for secure computation from ANSI-C. Using advanced optimization techniques that have been adapted to the MPC cost model, our compiler CBMC-GC outperforms state-of-the-art compilers in circuit size for various non-trivial applications, ranging from floating point operations to end-user applications as privacy preserving location aware scheduling.

For future work we plan to achieve a fully automatized decomposition on the source code level to enhance the scalability of CBMC-GC. Moreover, we plan to support built-in floating point operations and to investigate the integration of oblivious data structures in CBMC-GC, e.g., ORAM [21].

Acknowledgements We thank all anonymous reviewers for their helpful and constructive comments. 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". The initial idea behind CBMC-GC, i.e., using a bounded model checker for high-level synthesis in the context of MPC, was coined in a very fruitful discussion with Helmut Veith over a cup of coffee in a Wiener Kaffeehaus (typical Viennese coffee house).

A Code example: Hamming distance computation

The Hamming distance between two bit strings is the number of pairwise different bits. This number can be computed by XOR-ing the input bit strings and then counting the number of bits. An exemplary implementation is given in Listing 3 that computes the distance between two bit-strings of length 160 bits, which are split over five unsigned integers. In Line 7 the number of ones in a string of 32 bits is computed. This task is also known as population count. In the following paragraphs we describe three different implementations.

Listing 3 Hamming distance computation between two bit strings

```
1  #define N 5
2  void hamming() {
3    unsigned INPUT_A_x[N];
4    unsigned INPUT_B_y[N];
5    unsigned res = 0;
6    for(int i = 0; i < N; i++) {
7      res += count_naive32(INPUT_A_x[i]^INPUT_B_y[i]);
8    }
9    unsigned OUTPUT_res = res;
10 }
```

The first implementation is given in Listing 4. In this naïve approach, each bit is extracted using bit shifts and the logical AND operator `&` before being aggregated.

Listing 4 Counting bits, naïve approach

```
1  unsigned char count_naive32(unsigned y) {
2      unsigned char m = 0;
3      for(unsigned i = 0; i < 32; i++) {
4        m += (y & (1 << i)) >> i;
5      }
6      return m;
7  }
```

A variant of this implementation is given in Listing 5. Here, the bit string of length 32 is first split into chunks of 8 bits (`unsigned char`). The ones set in each chunk are then counted as described above.

Listing 5 Counting bits over unsigned chars

```
1  unsigned char count_naive8(unsigned char c) {
2      unsigned char m = 0;
3      for(int i = 0; i < 8; i++) {
4        m += (c & (1 << i)) >> i;
5      }
6      return m;
7  }
8
9  unsigned char count_tree32(unsigned y) {
10     unsigned char m0 = y & 0xFF;
11     unsigned char m1 = (y & 0xFF00) >> 8;
12     unsigned char m2 = (y & 0xFF0000) >> 16;
13     unsigned char m3 = (y & 0xFF000000) >> 24;
14     return count_naive8(m0) + count_naive8(m1) + \
15            count_naive8(m2) + count_naive8(m3);
16 }
```

Finally, in Listing 6 the best known implementation optimized for a CPU with 32 bit registers and slow multiplication is given. This implementation uses only 14 instructions.

Listing 6 Counting bits, optimized for a 32 bit CPU

```
1  unsigned count_reg32(unsigned y) {
2      unsigned x = y - ((y >> 1) & 0x55555555);
3      x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
4      x = (x + (x >> 4)) & 0x0f0f0f0f;
5      x += x >>  8;
6      x += x >> 16;
7      return x;
8  }
```

References

1. Berkeley logic synthesis and verification group, abc: a system for sequential synthesis and verification, release 30916. http://www.eecs.berkeley.edu/~alanmi/abc/
2. Bellare M, Hoang VT, Keelveedhi S, Rogaway P (2013) Efficient garbling from a fixed-key blockcipher. In: IEEE S&P
3. Bilogrevic I, Jadliwala M, Hubaux J, Aad I, Niemi V (2011) Privacy-preserving activity scheduling on mobile devices. In: ACM CODASPY
4. Bjesse P, Borälv A (2004) Dag-aware circuit compression for formal verification. In: ICCAD
5. Bogdanov D, Laur S, Willemson J (2008) Sharemind: a framework for fast privacy-preserving computations. In: ESORICS
6. Bogetoft P, Christensen DL, Damgård I, Geisler M, Jakobsen T, Krøigaard M, Nielsen JD, Nielsen JB, Nielsen K, Pagter J et al (2009) Secure multiparty computation goes live. In: FC
7. Buchfuhrer D, Umans C (2011) The complexity of Boolean formula minimization. J. Comput. Syst. Sci. 77(1):142–153
8. Buescher N, Holzer A, Weber A, Katzenbeisser S (2016) Compiling low depth circuits for practical secure computation. In: ESORICS
9. Buescher N, Kretzmer D, Jindal A, Stefan K (2016) Scalable secure computation from ansi-c. In: IEEE WIFS
10. Büscher N, Katzenbeisser S (2015) Faster secure computation through automatic parallelization. In: USENIX Security
11. Clarke EM, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: TACAS
12. Clarke EM, Kroening D, Yorav K (2003) Behavioral consistency of C and verilog programs using bounded model checking. In: DAC
13. Courtois N, Hulme D, Mourouzis T (2011) Solving circuit optimisation problems in cryptography and cryptanalysi. IACR cryptology ePrint archive
14. Damgård I, Pastro V, Smart NP, Zakarias S (2012) Multiparty computation from somewhat homomorphic encryption. In: CRYPTO
15. Darringer JA, Joyner WH, Berman CL, Trevillyan L (1981) Logic synthesis through local transformations. IBM J Res Dev 25:272–280
16. Demmler D, Dessouky G, Koushanfar F, Sadeghi AR, Schneider T, Zeitouni S (2015) Automated synthesis of optimized circuits for secure computation. In: ACM CCS
17. Demmler D, Schneider T, Zohner M (2015) ABY—a framework for efficient mixed-protocol secure two-party computation. In: NDSS
18. Erkin Z, Franz M, Guajardo J, Katzenbeisser S, Lagendijk I, Toft T (2009) Privacy-preserving face recognition. In: PETS
19. Franz M, Holzer A, Katzenbeisser S, Schallhart C, Veith H (2014) CBMC-GC: an ANSI C compiler for secure two-party computations. In: Compiler construction CC
20. Goldreich O, Micali S, Wigderson A (1987) How to play any mental game or a completeness theorem for protocols with honest majority. In: ACM STOC
21. Goldreich O, Ostrovsky R (1996) Software protection and simulation on oblivious rams. J ACM 43(3):431–473
22. Goudarzi D, Rivain M (2016) On the multiplicative complexity of boolean functions and bitsliced higher-order masking. In: CHES

23. Henecka W, Kögl S, Sadeghi AR, Schneider T, Wehrenberg I (2010) TASTY: tool for automating secure two-party computations. In: ACM CCS
24. Holzer A, Franz M, Katzenbeisser S, Veith H (2012) Secure two-party computations in ANSI C. In: ACM CCS
25. Kolesnikov V, Sadeghi AR, Schneider T (2009) Improved garbled circuit building blocks and applications to auctions and computing minima. In: CANS
26. Kolesnikov V, Schneider T (2008) Improved garbled circuit: free XOR gates and applications. In: ICALP
27. Kreuter B, Shelat A, Mood B, Butler K (2013) PCF: a portable circuit format for scalable two-party secure computation. In: USENIX security
28. Kreuter B, Shelat A, Shen C (2012) Billion-gate secure computation with malicious adversaries. In: USENIX security
29. Kuehlmann A (2004) Dynamic transition relation simplification for bounded property checking. In: IEEE ICCAD
30. Larraia E, Orsini E, Smart NP (2014) Dishonest majority multi-party computation for binary circuits. In: CRYPTO
31. Liu C, Huang Y, Shi E, Katz J, Hicks MW (2014) Automating efficient RAM-model secure computation. In: IEEE S&P
32. Liu C, Wang XS, Nayak K, Huang Y, Shi E (2015) ObliVM: a programming framework for secure computation. In: IEEE S&P
33. Malkhi D, Nisan N, Pinkas B, Sella Y (2004) Fairplay - secure two-party computation system. In: USENIX Security
34. Mishchenko A, Chatterjee S, Brayton R, Een N (2006) Improvements to combinational equivalence checking. In: IEEE ICCAD
35. Mishchenko A, Chatterjee S, Brayton RK (2006) Dag-aware AIG rewriting a fresh look at combinational logic synthesis. In: DAC
36. Mood B, Gupta D, Carter H, Butler K, Traynor P (2016) Frigate: a validated, extensible, and efficient compiler and interpreter for secure computation. In: IEEE Euro S&P
37. Mood B, Letaw L, Butler K (2012) Memory-efficient garbled circuit generation for mobile devices. In: FC
38. Nielsen JB, Nordholt PS, Orlandi C, Burra SS (2012) A new approach to practical active-secure two-party computation. In: CRYPTO
39. Robertson JE (1958) A new class of digital division methods. IRE Trans Electron Comput 3:218–222
40. Schneider T, Zohner M (2013) GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In: FC
41. Schnorr CP (1974) Zwei lineare untere Schranken für die Komplexität Boolescher Funktionen. Computing 13:155–171
42. Schröpfer A, Kerschbaum F, Müller G (2011) L1—an intermediate language for mixed-protocol secure computation. In: COMPSAC
43. Songhori EM, Hussain SU, Sadeghi A, Schneider T, Koushanfar F (2015) Tinygarble: Highly compressed and scalable sequential garbled circuits. In: IEEE S&P
44. Turan MS, Peralta R (2014) The multiplicative complexity of boolean functions on four and five variables. In: LightSec
45. Yao ACC (1982) Protocols for secure computations (extended abstract). In: IEEE FOCS
46. Yao ACC (1986) How to generate and exchange secrets (extended abstract). In: IEEE FOCS
47. Zahur S, Evans D (2015) Obliv-c: a language for extensible data-oblivious computation. IACR cryptology ePrint archive