LLVM-Based Circuit Compilation for Practical Secure Computation

Tim Heldmann, Thomas Schneider, Oleksandr Tkachenko, Christian Weinert(✉), and Hossein Yalame

Technical University of Darmstadt, Darmstadt, Germany
tim.heldmann@stud.tu-darmstadt.de,
{schneider,tkachenko,weinert,yalame}@encrypto.cs.tu-darmstadt.de

Abstract. Multi-party computation (MPC) allows two or more parties to jointly and securely compute functions over private inputs. Cryptographic protocols that realize MPC require functions to be expressed as Boolean or arithmetic circuits. Deriving such circuits is either done manually, or with hardware synthesis tools and specialized MPC compilers. Unfortunately, such existing tools compile only from a single front-end language and neglect decades of research for optimizing regular compilers.

In this paper, we make MPC practical for developers by automating circuit compilation based on the compiler toolchain LLVM. For this, we develop an LLVM optimizer suite consisting of multiple transform passes that operate on the LLVM intermediate representation (IR) and gradually lower functions to circuit level. Our approach supports various front-end languages (currently C, C++, and Fortran) and takes advantage of powerful source code optimizations built into LLVM. We furthermore make sure to produce circuits that are optimized for MPC, and even offer fully automated post-processing for efficient post-quantum MPC.

We empirically measure the quality of our compilation results and compare them to the state-of-the-art specialized MPC compiler HyCC (Büscher et al. CCS'2018). For all benchmarked HyCC example applications (e.g., biomatch and linear equation solving), our highly generalizable approach achieves similar quality in terms of gate count and composition.

Keywords: MPC · Circuit compilation · LLVM · Hardware synthesis

1 Introduction

Multi-party computation (MPC) allows two or more parties to jointly compute functions over their respective inputs, while the privacy of the inputs is ensured and nothing but the functions' output is revealed. First cryptographic protocols to realize MPC were proposed already in the 1980's by Yao [63] and Goldreich, Micali, and Widgderson [23]. However, it took until 2004 for MPC to be implemented efficiently [41] and see adoption for privacy-preserving applications, e.g., private set intersection [28,49] and privacy-preserving machine learning (PPML) [7,20,46,48].

K. Sako and N. O. Tippenhauer (Eds.): ACNS 2021, LNCS 12727, pp. 99–121, 2021.
https://doi.org/10.1007/978-3-030-78375-4_5

The widespread adoption of MPC is unfortunately compromised by the requirement to implement the functions to be computed as Boolean or arithmetic circuits. This process is tedious and error-prone when done by hand, and additionally requires an extensive understanding of the underlying cryptographic protocols to build circuits that can be evaluated efficiently.

To address this issue, previous works tried to develop toolchains for automatic compilation of high-level code to circuit representations. Most notably, the specialized MPC compiler HyCC [13] compiles ANSI C to optimized Boolean and arithmetic circuits. However, works like HyCC allow compilation only from a very limited subset of a *single front-end language* and more importantly neglect decades of research that went into building and optimizing conventional compilers like GCC and the versatile LLVM toolchain [38], which we leverage in this paper.

Other works like TinyGarble [17,56] rely on logic synthesis tools (e.g., Synopsis Design Compiler [57] or Yosys-ABC [1,62]) to generate net lists of Boolean circuits. However, logic synthesis tools require knowledge of hardware description languages like Verilog or VHDL, and were built to optimize electronic circuits in terms of clock cycle usage, routing, and placing [58]. To match the cost metrics relevant for efficient MPC and restrict circuit generation to the supported basic gate types, it is nevertheless possible to re-purpose such tools by providing custom constraints and technology libraries.

Our Contributions. In this paper, we aim at making MPC practical for software developers by automating circuit compilation based on the compiler toolchain LLVM [38]. For this, we design and implement an LLVM optimizer suite consisting of multiple transform passes that gradually lower functions to circuit level. For example, the passes remove conditional branching and array accesses, eliminate constant logic, and replace low-level instructions with optimal building blocks that we obtain via logic synthesis tools.

Our LLVM optimizer suite operates entirely on the level of the LLVM intermediate representation (IR). Thus, we naturally facilitate compilation from numerous existing front-ends for a wide range of programming languages. We currently support a subset of C, C++, and Fortran, and give a road map for extensions to additional high-level languages like Rust. Furthermore, our approach takes advantage of powerful source code optimizations built into LLVM, which can swiftly deliver significant improvements in terms of resulting circuit sizes and therefore MPC performance.

To bridge the gap from LLVM-IR to MPC frameworks, we provide a converter to the widely known Bristol circuit format [59] that can be evaluated, e.g., by ABY [18] and MOTION [11]. Instead of converting LLVM-IR to Bristol, it is also possible to further compile via LLVM back-ends to a conventional architecture like x86. Besides testing for functional correctness, such circuit-based binaries have further applications in mitigating cache side-channel attacks [42].

Even though we construct our circuits from optimal building blocks, the assembled result might not be overall optimal. Therefore, to compete with sophisticated circuit-level optimizations delivered by specialized MPC compilers like HyCC [13], we add fully automated post-processing. For this, we convert Bristol to Verilog net lists, apply optimization passes of synthesis tools [1,57,62], and convert back to Bristol. By defining different constraint sets and technology libraries, we not only support size-optimized Boolean circuits considering the free-XOR [35] and half-gates technique [65], but also cost metrics for post-quantum secure MPC [12]. Our approach is generic and can easily be adjusted for further cost metrics that might be of interest, e.g., garbling under standard assumptions [24].

Finally, we empirically measure the quality of our compilation results and compare them to the specialized MPC compiler HyCC [13]. For this, we benchmark example applications similar as in the HyCC repository (e.g., biomatch and linear equation solving via Gaussian elimination) as well as additional implementations. Our LLVM-based approach achieves similar or even better quality in terms of gate count and composition, while also providing a richer feature set and the benefits of extensibility to other high-level languages as well as MPC cost metrics.

In short, we summarize our contributions as follows:

1. Design and implementation of an LLVM-based toolchain for circuit compilation from various high-level languages (currently a subset of C, C++, and Fortran) publicly available at https://encrypto.de/code/LLVM.
2. Fully automated post-processing of Bristol circuits via logic synthesis tools to optimize for (post-quantum) MPC-relevant cost metrics [12,35,65].
3. Empirical performance evaluation showing similar or better quality compared to the specialized MPC compiler HyCC [13].

Outline. We introduce the necessary MPC as well as compiler background in Sect. 2 and discuss related works in Sect. 3. In Sect. 4, we present our LLVM-based circuit compilation toolchain and propose fully automated post-processing in Sect. 5. We evaluate our results in Sect. 6 before concluding with a comprehensive outlook in Sect. 7.

2 Preliminaries

We introduce the necessary background on MPC (cf. Sect. 2.1), LLVM (cf. Sect. 2.2), and logic synthesis (cf. Sect. 2.3).

2.1 Multi-party Computation

Multi-party computation (MPC) allows N mutually distrusting parties to jointly compute functions over their respective inputs. In addition to the correctness of the function output, MPC guarantees privacy of the inputs as nothing but the function output is revealed. Two seminal cryptographic protocols that realize MPC and are still highly relevant are Yao's garbled circuits [63] and the protocol by Goldreich, Micali, and Wigderson (GMW) [23]. The work by Beaver, Micali, and Rogaway (BMR) [5] extends the idea of Yao from the two- to the multi-party case. All these protocols obliviously evaluate Boolean circuit representations of the desired functionality. While Yao/BMR evaluate the circuit in a constant-round protocol, the number of communication rounds for GMW depends on the multiplicative depth of the circuit. In this work, we for now focus on the compilation of size-optimized Boolean circuits for the first approach (Yao/BMR), while an extension to depth-optimized circuits for GMW is straightforward.

Yao's Garbled Circuits/BMR. In Yao's protocol [63], one party (the garbler) generates a garbled circuit corresponding to a Boolean circuit by encrypting the truth table of each gate. The other party (the evaluator) then evaluates (decrypts) the circuit received from the garbler. Transferring the required keys corresponding to the chosen inputs is done via a cryptographic protocol called oblivious transfer (OT) [30]. Today's most efficient solution for garbled circuits is a combination of free-XOR [35] and half-gates [65]. With these optimizations, each AND gate requires the transfer of two ciphertexts, whereas XOR gates are essentially free (no communication necessary). BMR [5] is an extension of Yao's protocol to the multi-party case, where all parties garble and evaluate the circuit such that no subset of parties learns anything about the intermediate values.

Post Quantum Yao. A post-quantum version of Yao's protocol was recently proposed in [12], where the adversary has access to a quantum computer. The authors proved Yao's protocol in the quantum world under the assumption that the used encryption scheme is PQ-IND-CPA. They assume the quantum adversary has only access to a quantum random oracle and does not make queries to the encryption oracle in superposition. Since the free-XOR optimization [35] was established under a weaker assumption, it cannot be applied in the quantum world, and thus the cost of XOR is the same as for AND gates.

Bristol Circuit Format. Circuit descriptions specify either the transfer of data on register level (RTL) or a list of gates, wires, and their connections (net list). The most commonly used circuit format for MPC is the Bristol format [3,59], which is a plain text net list format. Each line in a Bristol file represents a single INV, AND or XOR gate. Each gate can have one or two inputs and has exactly one output, all of which are specified via IDs. Every output ID is unique, in static single assignment (SSA) form, and all IDs are in ascending order. The number of input as well as output wires is specified in the header of the file.

Frameworks. There exist many MPC frameworks [52] implementing different protocols and considering different adversaries w.r.t. the ratio of honest/dishonest parties (honest/dishonest majority or full threshold) and the behaviour (semi-honest, covert, malicious). MPC frameworks take either a (subset) of a high-level language language, a domain specific programming language, or a net list as input. Our approach generates circuits in the Bristol net list format, which we empirically evaluate using the state-of-the-art framework MOTION [11].

2.2 LLVM

LLVM started as a research project in 2004 with the goal to enable lifelong program analysis and transformation for arbitrary software [38]. While originally being an acronym for low-level virtual machine, the LLVM project now spans different sub projects including, e.g., the specification for the LLVM intermediate representation (LLVM-IR), the LLVM-IR optimizer, and the Clang C/C++ front-end. The intended LLVM workflow to generate binaries is as follows:

1. Compile source code with a compatible front-end to LLVM-IR.
2. Use the LLVM optimizer to optimize the LLVM-IR.
3. Use a back-end to compile the optimized LLVM-IR to an executable binary.

Due to the strict separation between the toolchain steps, LLVM is highly extensible. Especially the default optimization passes can be extended with custom language- or hardware-specific passes. In the following, we introduce the LLVM-IR and optimization passes in more detail.

LLVM Intermediate Representation (LLVM-IR). The LLVM-IR is the connection between the front- and back-ends. It exists in three forms: as (i) in-memory IR used by the compiler, (ii) bytecode for just-in-time compilers, and (iii) a human readable assembly language. While all of them are equivalent, and can be converted losslessly, all further mentions of LLVM-IR refer to the assembly language.

LLVM-IR is a static single assignment (SSA)-based, strictly typed representation of the translated source code. For translating to LLVM-IR, features common in non-SSA-based languages must to be transformed. One typical problem here is the differing state of variables depending on previously run parts of the program, as occurs with branching. The solution for SSA restrictions in LLVM-IR are so-called φ-nodes. Such nodes are instructions that evaluate differently depending on which part of the program was executed last. The same problem occurs when dealing with loops, which often can also be resolved via loop unrolling.

LLVM Optimizer. The LLVM optimizer is intended to perform LLVM-IR to LLVM-IR transformations. It utilizes optimizer passes that run on specific parts of the LLVM-IR. The optimizer comes with a set of language independent optimizations [40], but can be extended with additional passes. Passes can be categorized as analysis and transform passes. Analysis passes generate additional information about LLVM-IR code without any modifications. Transform passes, on the other hand, are allowed to modify the LLVM-IR, possibly invalidating previously run analyses in the process.

2.3 Logic Synthesis for MPC Circuit Compilation

Logic synthesis tools take a function description in a hardware description language (HDL) such as Verilog or VHDL as input, and transform it to the respective target technologies, e.g., look-up tables (LUTs) for field programmable gate arrays (FPGAs) or Boolean gates for application-specific integrated circuits (ASICs). Since creating hand-optimized Boolean circuits for MPC is an error-prone and time-consuming task, it is a promising and natural approach to utilize existing logic synthesis tools. However, software developers are rarely familiar with HDLs and re-purposing such tools for performing MPC-specific optimizations requires the development of custom ASIC technology libraries. In this work, we utilize the open-source Yosys-ABC synthesis tool [1,62]. In contrast to previous works [17,19], we not only create optimized building blocks, but provide a fully automated compilation and MPC-optimization workflow from several high-level programming languages. Furthermore, we are the first to develop a custom ASIC technology library for post-quantum MPC [12].

3 Related Work

Our LLVM-based circuit compilation toolchain allows software developers to use multiple different general-purpose programming languages to produce circuits for MPC. To the best of our knowledge, we are the first to utilize LLVM for such an endeavour. The recently initiated CIRCT project [39] instead aims at replacing HDLs like Verilog with IR as a portable format between hardware design tools and utilizes the LLVM infrastructure for offering transform passes between different abstractions as well as architectures. On the other hand, there exist multiple tools that allow developers to generate circuits from a *single* high-level or domain-specific language. In the following, we first give an overview of and comparison between these tools. Then, we briefly review MPC frameworks that evaluate such generated circuits or can be programmed with custom MPC-specific languages, and choose one of them for benchmarking circuits generated via our LLVM toolchain (cf. Sect. 6.2).

3.1 Circuit Generation

The following approaches generate circuits from high-level code, but like our LLVM-based circuit compilation approach abstract away the circuit evaluation.

Dedicated Compilers for MPC. TinyGarble [17,56] uses logic synthesis to generate efficient Boolean circuits for MPC from Verilog net lists. However, since TinyGarble's optimization process requires a Verilog net list, it is reliant on an external high-level synthesis (HLS) tool to compile high-level code. Our approach is designed to be compatible with various LLVM front-ends for compiling high-level languages, and focuses on the generation of circuits from LLVM-IR. Since many programmers are unfamiliar with HDLs, we target a much larger community of software developers.

CBMC-GC is an extension of CBMC [15]. It converts ANSI C to a Boolean circuit [27] and proves that the generated circuit is equivalent to the input program. While being compatible with ANSI C, CBMC-GC has limitations in terms of variable naming and is restricted to inputs from two parties. Not only is our approach input and variable name agnostic, as the circuit's inputs are generated depending on the function's signature, but it is also not limited to ANSI C. Furthermore, we make use of the source code optimization suite of LLVM. Starting circuit synthesis from optimized code can be very advantageous (cf. Sect. 4.1).

Although our implementation does not have a formal verification proof, the correctness of our compiler can be tested easily, e.g., by running a test suite after compiling the transformed LLVM-IR code via LLVM back-ends to x86 binaries.

HyCC [13] is a compiler that extends CBMC-GC for hybrid circuits that contain Boolean and arithmetic parts for efficient mixed-protocol evaluation. For this, it partitions an ANSI C program into modules using a heuristic and then assigns the most suitable MPC protocol in terms of runtime and/or communication. Furthermore, HyCC applies sophisticated circuit-level optimizations to increase efficiency. Currently, our toolchain generates only Boolean circuits and does not support automatic protocol selection. Manual switching of protocols can be implemented in the "gateify" pass (cf. Sect. 4.3), and automated protocol selection can be added by developing an optimization pass based on the work of [31] (cf. Sect. 7). Instead of transform passes that perform circuit-level optimizations, we optimize circuits via post-processing by utilizing logic synthesis tools (cf. Sect. 5).

The portable circuit format (PCF) compiler [36] generates circuits in the PCF format using bytecode generated by LCC [22] from ANSI C. This allows for optimizations on the bytecode level such as dead gate removal and constant propagation. Then, it uses an internal language to translate instructions from the bytecode to a Boolean circuit. In comparison to net list formats, PCF features loops and recursion, removing the need for recursive function inlining as well as loop unrolling. We also make use of bytecode-level optimizations and additionally feature circuit-level post-processing. In contrast to our toolchain, PCF relies on the LCC compiler, which supports only ANSI C.

High-Level Synthesis. High-level synthesis (HLS) is an approach for designing circuits in a high-level language by specifying the desired behavior. The exact translation into a chip design is controlled by a compiler. HLS systems usually require expert knowledge, as they rely on domain-specific programming languages like Verilog [16,60] instead of C, C++, or Fortran.

Another difference between HLS tools and our approach is that their goal is to create electrical circuits, which have different cost metrics than MPC. In HLS, much thought is given to routing and placement algorithms, which is of limited use for optimizing circuits for MPC. Being designed with MPC applications in mind, our approach leads to better extensibility and less overhead.

Domain-Specific Languages. PAL [45] compiles a domain-specific language (DSL) into a size-optimized Boolean circuit. The scalable KSS compiler [37] generates Boolean circuits from a DSL by employing a constant propagation optimization method. SMCL [47], L1 [55], and Wysteria [50] are custom high-level languages for describing MPC that support the combination of different MPC protocols. Wysteria additionally provides a tool for circuit compilation. However, it is based on functional programming and therefore tedious to learn by developers who are trained in imperative programming.

Obliv-C [64] and EzPC [14] extend a general-purpose programming language with MPC-specific functionality descriptions, e.g., secret/public variables and oblivious `if`, and automatically compile executables. Also, EzPC supports automatic protocol assignment for mixed-protocol computation and uses the ABY framework [18] to compile executable binaries.

3.2 MPC Frameworks for Circuit Evaluation

There exist many MPC frameworks that allow users to run MPC protocols. They take as input a circuit description or a domain-specific language for MPC. For a comprehensive overview, we refer to [25]. Here, we briefly describe popular MPC frameworks and justify the choice for our benchmarks (cf. Sect. 6.2).

Fairplay [41] was the first framework for secure two-party computation and FairplayMP [6] is its extension to multiple parties. Both use the secure function description language (SFDL) to describe functions and convert to the secure hardware description language (SHDL). Sharemind [8] implements 3-party additive secret sharing with honest majority and is a proprietary software programmed with SecreC. TASTY [26] is a framework for two parties that allows to mix garbled circuits with homomorphic encryption for applications implemented using a subset of Python. FastGC [29] is a two-party framework that is implemented in Java and uses garbled circuits. It allows gate-level pipelining of the circuit, which reduces the memory overhead. Frigate [44] consists of an efficient compiler and an interpreter. The compiler takes a custom C-style language and ensures the correctness of the generated circuits.

ABY [18] and ABY3 [43] are mixed-protocol MPC frameworks written in C++ for two- and three-party computation, respectively. FRESCO is a Java MPC framework that implements additive secret sharing schemes. EMP toolkit [61] implements a few MPC protocols and oblivious transfer in C++, and provides a low-level API for cryptographic primitives. PICCO [66] compiles an input description written in a custom extension of C to C and runs it using N-party threshold MPC. JIFF [9] is a framework for information-theoretically secure MPC written in JavaScript, which allows to use it in web applications. MPyC [54] is a Python framework for N-party computation based on secret sharing protocols. MP-SPDZ [33] implements multiple MPC protocols and cryptographic primitives in different security models. SCALE-MAMBA [2] is a framework for N-party mixed-protocol maliciously secure MPC that compiles a Python-like language to bytecode that is parsed by a "virtual machine" that runs MPC protocols.

In this work, we use the MOTION framework [11] for benchmarking the circuits generated by our LLVM toolchain (cf. Sect. 6.2). MOTION is an N-party mixed-protocol MPC framework implemented as a C++ library. It supports the BMR [5] as well as the GMW protocol [23] (cf. Sect. 2.1). It guarantees semi-honest security against all but one corrupted parties (full threshold). MOTION provides a user-friendly API for evaluating circuits in the Bristol format [3,59], which is also the format our toolchain produces.

4 LLVM-Based Circuit Compilation

We now present our LLVM-based circuit compilation approach. For this, we first give an intuitive overview of our optimizer suite of LLVM transform passes that ultimately compile high-level programs to Bristol circuit representations. Afterwards, each transform pass is described in detail.

4.1 Overview

In this section, we give an intuition for our transformation pipeline that consists of a suite of LLVM-IR transform passes.

Using a compatible LLVM front-end (e.g., Clang for C and C++, or Flang for Fortran), we first compile the given code to LLVM-IR code. At this point, we can apply all source code optimizations shipped by LLVM and additionally perform loop unrolling. For example, the control flow simplifying pass "simplify-cfg" removes unused branches and pre-computes constant branching logic. The instruction combine pass "instcombine" simplifies Boolean or arithmetic instructions. For example, the function `return b^(a*b/a);` operating on integers a and b always returns 0. The LLVM optimizer detects this in 117 ms , while HyCC [13] takes 25 s to generate a circuit with 6416 gates. The output of this stage is the basis for circuit generation and will be referred to as the "base function".

As branching is not trivially supported on circuit level, we then proceed with eliminating all branches. This is done by our so-called "phi remove" pass, described in Sect. 4.2. For this, we have to inline all basic blocks (i.e., blocks of sequentially executed instructions), and swap φ-nodes to select instructions (the LLVM-IR representation of a ternary expression).

Now that all code is contained in one basic block, our "gateify" pass, described in 4.3, replaces all instructions with "circuit-like" functions. A circuit-like function is a function that first disassembles the inputs into single bits, evaluates the function exactly as a circuit consisting of primitive gates, and reassembles the result to the required datatype.

Next in line is the "array to multiplexer" pass. Its basic concept is similar to the gateify pass as it swaps arrays for calls to functions that behave like multiplexers and thus enable *oblivious* data accesses on circuit level. The exact differences are described in Sect. 4.4.

Since at this point the program's code is distributed over various external functions, we now apply the "merge function" pass (cf. Sect. 4.5). It takes all the different external function calls, merges their content into a single function, and wires the outputs to inputs accordingly. The resulting function behaves very similar to a circuit consisting of primitive gates that represents the same function.

To further reduce the size of the generated function, and therefore the circuit, a final pass is applied. The "constant logic elimination" pass, described in Sect. 4.6, simplifies logic instructions, by either pre-computing the result in case of two constant operands, or passing the corresponding value or constant in case of one constant operand, or both operands being the same. Lastly, if the result of an instruction is never used, it is removed entirely.

The LLVM-IR code consisting of only gate-like instructions can then be trivially converted to the Bristol format, using our LLVM-to-Bristol converter (cf. Sect. 4.7). Then, we convert Bristol to a Verilog net list and apply our post-processing using logic synthesis tools (cf. Sect. 5). This results in an even smaller circuit that we convert back to Bristol. The latter constitutes the final result of our toolchain.

4.2 Phi Remove Pass

The LLVM-IR is in static single assignment (SSA) form (cf. Sect. 2.2). Therefore, conditional branching is difficult to represent, as variables can have different values depending on COMPthe evaluation of condition statements. The LLVM-IR solution for this problem are φ-nodes that take different values depending on the COMPpreviously evaluated basic block. While the basic premise of SSA holds true for circuits, φ-nodes must be replaced. Instead of calculating only one path depending on the branch condition and generating the value in the φ-node depending on the source basic block, we evaluate both branches regardless of the condition and replace φ-nodes with a multiplexer. The selection bit of the multiplexer is the result of the branching condition.

This approach works flawlessly for two-way branching, e.g., regular if/else instructions. The LLVM-IR specification, however, allows for an arbitrary number of values to be taken by a φ-node, as a basic block can be branched to by any number of other basic blocks. This can be especially useful when trying to represent switch/case instructions. It requires analysis of the conditions leading to the branch, as well as a multiplexer tree instead of a single multiplexer.

To summarize, the phi remove pass identifies two-way branches, recursively descends the basic blocks, replaces φ-nodes with ternary select instructions (which are handled later), and splices the instruction lists of the basic block together with the goal to ultimately achieve a function that only has one basic block.

4.3 Gateify Pass

The "gateify" pass iterates through every function of the module and identifies supported instructions. Once such an instruction is found, the pass creates a new "circuit-like" function with the same behavior. The exact instructions of this new function are defined by our building blocks. In Appendix A we elaborate on how we utilized hardware synthesis tools similar to [17,56] in order to obtain optimal circuits for primitive instructions that can serve as building blocks.

As Boolean MPC circuits rely on bitwise operations while LLVM-IR uses static types larger than one bit, it is necessary to disassemble the static compound types like `i8` or `i32` into 8 or 32 `i1` types, respectively. This disassembly process is done as a prephase inside the new function. The computation of the result is then done by applying the instructions specified in the circuit description of the building block. Once all circuit instructions have been copied, the result is then given as `i1` types that have to be reassembled to a compound type. This reassembly process is the postphase of the building block.

The newly created function signature and return value are matched to be equal to the operands and result of the replaced instruction. The original instruction is then deleted and all uses of the calculated result are replaced with the result returned by the newly created function. Internally, the new function is assigned the attribute "gate" to mark it as a circuit-like function. This will be important to identify mergeable functions later.

4.4 Array to Multiplexer Pass

The "array to multiplexer" pass is required for source languages that support arrays as a construct and corresponding front-ends use the LLVM-IR array construction to represent them. For example, LLVM-IR code generated by Clang for C and C++ will use the LLVM-IR array construct. On the other hand, Fortran instead of arrays has the concept of multidimensional fields, which behave similar but are not represented as arrays in LLVM-IR.

The pass first analyzes the array usage of a given function. During this analysis, all stores to constant positions are mapped out. If the same position is written multiple times, the updated values are saved as well.

However, if a value should be stored to a position unknown at compile time, every single position could be affected. To support stores to variable positions, every position of the array can be updated with the result of a ternary instruction, similar to this: `i == unkownPos ? newValue : array[i]`. This updates every position with its own value, except the one position where the condition evaluates to true, leading to this position being updated to the new value.

Once all stores are mapped out, the analysis result is used to replace all reads from a constant position with the value that is at the corresponding position at the time the read occurs. This value is the last update that happened to that position before the read occurred. If a value is loaded from a position that is not known during the compilation, the whole array is given to a multiplexer tree, with the position disassembled to bits as the decision bits of the multiplexer tree in the corresponding layer.

4.5 Merge Pass

The "merge" pass creates a single circuit from all circuit-like functions. It first creates its own prephase, disassembling every parameter of the base function to provide as primitives for the merged function. Then, it clones the instructions of all circuit-like functions in the new merged function, excluding the pre- and postphase. As the instructions of the base function were topologically sorted, it is guaranteed that the first instruction to clone will only access primitives or constants. Following functions will either reference primitives or intermediate results that have already been disassembled.

After successfully cloning every instruction of a building block, it is necessary to map the cloned instructions such that their references match the context of the function they got cloned into. The output bits of every instruction are saved and mapped to the corresponding inputs of any instruction that references them. Once the return statement is reached, a postphase is added that reassembles the output bits to match the return type of the base function.

4.6 Constant Logic Elimination Pass

The "constant logic elimination" pass cleans up the merged function. It iterates through every instruction in the merged function. Due to the gateify pass, every instruction is either an AND, XOR, or INV instruction. In case both operands for an AND or XOR instruction are constants, the result is computed, and every instruction referencing the result updated to use the pre-computed value instead. The original instruction is then removed. In case one operand is a constant, a lookup table determines whether to keep the instruction or replace it with a constant or the operand. Once all logic gates with constant operands are eliminated, a last pass is done to remove instructions with unused results.

4.7 Export to Bristol Format

After all passes are executed, the resulting LLVM-IR file contains a fully merged function with exactly one basic block. This file can then be passed to our "LLVM-IR to Bristol" converter. The converter will skip past the disassembly phase and locate the gate operations. Each line is then converted to a line in the Bristol file, until the end of the gate section is reached. As it is idiomatic to the Bristol format to state the amount of gates and wires in the beginning of the file, as well as the amount of parameters and the bit width of the result, they are calculated and prepended. Finally, the unique references between LLVM-IR instructions are mapped to wire identifiers in ascending order while ensuring that result bits are mapped to the highest numbered wire identifiers.

5 Post-processing Circuits for MPC

In Sect. 4, we described our LLVM-based circuit compilation approach that gradually lowers high-level implementations to circuit level. Specifically, our "gateify" transform pass (cf. Sect. 4.3) replaces all low-level LLVM-IR instructions with functionally equivalent building blocks. We designed these building blocks to be optimal according to MPC-relevant cost metrics, as was previously done in [17,19,32,56] (cf. Appendix A). However, we did not develop a transform pass that performs optimizations on the overall circuit, i.e., across building blocks. Therefore, our generated circuits are likely larger than those generated by specialized MPC compilers like HyCC [13] that include such optimizations, e.g., to replace highly redundant circuit parts.

To not reinvent the wheel, we instead propose to utilize HDL synthesis tool on the output of our LLVM-based circuit compilation for global optimizations. For this, we create a fully automated pipeline that first converts Bristol circuit descriptions to Verilog net lists, applies optimization passes of logic synthesis tools [1,57,62], and converts back to Bristol. By defining different constraint sets and technology libraries, we not only support size-optimized Boolean circuits considering the free-XOR [35] and half-gates technique [65], but also cost metrics for post-quantum secure MPC [12] (cf. Sect. 6). Additionally, this approach is generic and can easily be adjusted for further cost metrics that might be of interest, e.g., garbling under standard assumptions [24].

The conversion from Bristol to Verilog net lists is trivial due to their similarity in terms of structure and abstraction level. Yosys-ABC [1,62] then generates a net list output under synthesis objectives, which are provided by the developer to optimize the parameters like minimizing the delay or limiting the area of a synthesized circuit. We therefore develop customized technology libraries of basic gates, which include synthesis parameters like timing and area to guide the mapping. Concretely, we want to output a functionally equivalent yet optimized Boolean circuit net list consisting of only 2-input AND and XOR as well as INV gates. For the conversion back to Bristol, we utilize existing tooling from SCALE-MAMBA [2], which we extend to parse custom Verilog modules.

In the following, we detail our custom constraints and technology libraries. The performance (both in terms of runtime and circuit quality improvement) is evaluated in Sect. 7.

5.1 Customized Logic Synthesis for MPC

Regarding MPC protocols, we focus on Yao and BMR in this work (cf. Sect. 2.1), and therefore Boolean circuits. The relevant cost metric for both protocols is the multiplicative size, i.e., the number of AND gates in the circuit. In contrast, the cost of XOR evaluation is negligible [35]. We configure Yosys-ABC to minimize the multiplicative size by setting the XOR and INV gate area to 0 and for AND gates to a high non-zero value.

5.2 Customized Logic Synthesis for Post-quantum MPC

In the post-quantum setting, the previously discussed free-XOR optimization [35] is not applicable (cf. Sect. 2.1). Therefore, the relevant cost metric shifts from multiplicative size to the total gate count.

In order to meet our goal in post-quantum MPC, we design a customized library containing 2-input XOR as well as non-XOR gates. We set the area of all gates to an equal non-zero value and synthesize circuits considering area optimization as the main restriction. By doing so, we provide highly optimized circuits for post-quantum MPC that can moreover be conveniently compiled from various high-level languages.

6 Evaluation

We evaluate our LLVM-based approach to MPC circuit compilation in two aspects. First, we provide a complexity analysis of our transform passes. Then, we measure the quality of the generated circuits with respect to MPC cost metrics in comparison with HyCC [13] and provide concrete runtime as well as communication overheads when executing such circuits with the recent MPC framework MOTION [11]. The implementation of our toolchain is publicly available at https://encrypto.de/code/LLVM.

6.1 LLVM Transform Passes Complexity Analysis

The runtime of our transform passes depends on the number of instructions, basic blocks, and the size of the required building blocks. A summary of all complexities analyzing the worst case for each pass can be found in Table 1. Note that it is impossible for the worst case to occur simultaneously in all passes.

Let I be the number of instructions in the base function, B the number of basic blocks, N the size of the biggest utilized building block, and A the largest number of elements in an array.

Table 1. Complexity of our passes. I is the number of instructions, B the number of basic blocks, A the number of array slots, and N the size of the largest building block.

Pass	φ-Remove	Gateify	Array2MUX	Merge	C. Log. Elim.	**Total**
Complexity	$O(I+B)$	$O(I \cdot N)$	$O(I \cdot A)$	$O(I^2 \cdot N^2)$	$O(I \cdot N)$	$O(I^2 \cdot N^2)$

Phi Remove Pass. The phi remove pass's runtime mainly depends on the size and the number of basic blocks in the function. It recursively descends through all B basic blocks, replacing each φ-instruction with a ternary one. Once all the φ-instructions are replaced, the basic block merging is linear in complexity, as all instructions are saved in a doubly linked list, where we can splice the instructions of any basic block into any other basic block. Since in the worst case all I instructions are φ-instructions, this leads to a complexity of $O(I+B)$.

Gateify Pass. The gateify pass's runtime depends on the number of instructions and the size of the building blocks replacing them. Once an instruction has been replaced at least once, we can reference the building block for all identical instructions later on. But since we cannot universally assume duplicate instructions, the complexity class is $O(I \cdot N)$.

Array to Multiplexer Pass. The array to multiplexer pass iterates through all I instructions to identify `getelementptr`instructions. The modification step can then either forward constant reads with a complexity of $O(1)$, or create a multiplexer tree. For arrays with A elements, this tree has size $2^{\lceil \log_2(A) \rceil} - 1 \approx A$. The total complexity is therefore $O(I \cdot A)$.

Merge Pass. The merge gate function pass copies all instructions from the building blocks into a single function. While all other passes operate on the base function, this pass copies and re-maps every instruction in the building block functions. It can also not rely on the same splicing technique as the phi remove pass as the splice instruction is moving and not copying the instructions, and we might need to reference the building block again later. This leads to a complexity of $O(I^2 \cdot N^2)$, which makes the merge pass the bottleneck.

Constant Logic Elimination Pass. The constant logic elimination pass goes through all now $O(I \cdot N)$ instructions and pre-computes as well as propagates the logic if possible. Then, all unused computations are removed. This means $O(I \cdot N)$ instructions are inspected/modified.

6.2 LLVM Compilation Performance and Quality Analysis

We now measure the quality of the circuits generated by our LLVM-based toolchain (cf. Sect. 4) with respect to MPC cost metrics, and especially the benefits of our fully automated post-processing (cf. Sect. 5). Furthermore, we benchmark these circuits with the recent MPC framework MOTION [11] to analyze concrete runtime as well as communication overheads.

All these aspects we also compare to the HyCC compiler [13] to demonstrate that our very extensible approach can compete with or even outperform specialized MPC tools. Therefore, we mainly base our evaluation on applications written in C from the HyCC repository that we detail below.

Table 2. Compile time of different programs in seconds, compiled with HyCC [13] and LLVM, and with post-processing for minimizing the number of AND gates (LLVM$^+$) and the total number of gates for more efficient post-quantum MPC (LLVM PQ$^+$).

Program	HyCC [13]	LLVM	LLVM$^+$	LLVM PQ$^+$
Euclid	1.34	0.68	5.59	5.26
Dummy	2.20	1.32	10.73	10.88
Gauss	11.81	11.08	127.76	125.14
Biomatch	12.20	14.11	152.16	147.80

Benchmark Applications. The "Euclid" benchmark calculates the squared Euclidean distance between two points. This is a small and simple program, which shows basic translation capabilities.

The "biomatch" benchmark is similar to Euclid, but additionally calculates the square root of the result using Heron's method [21] with a cutoff at 20 iterations. This benchmark is used to show loop unroll handling and how highly repetitive functions can be greatly optimized with our post-processing approach.

The “Gauss” benchmark is a linear equation solver for up to 4 variables. It implements a forward elimination and backward substitution. This shows the difference if non-repetitive loops are unrolled and translated.

Additionally, we implement a “dummy” application that showcases as many supported features as possible in a comprehensive manner. The code for this application is attached in Appendix B.

Compile Time. In Table 2, we provide the runtime of the compilation for HyCC and our LLVM toolchain as well as the post-processing steps, measured on four cores of an Intel Xeon Gold 6144 CPU @ 3.50 GHz and 32 GB of RAM.

The compilation times of both approaches are comparable. For smaller applications (Euclid, dummy), LLVM is about twice as fast as HyCC. On the other hand, for a larger application (biomatch), HyCC is slightly faster. This is due to the comparatively high complexity of our merge pass (cf. Sect. 6.1) when handling large and redundant circuits, which is currently the bottleneck in our optimizer suite. Finally, we observe that the respective post-processing steps add a significant overhead of factor 10x on top of the basic compilation. However, we note that this is a one-time cost that occurs only before deployment when the development of an application is finalized.

Circuit Size and Composition. In Table 3, we show the circuit sizes and the composition of the compilation results. The basic compilation step with LLVM based on our transform passes already delivers circuits in the same order of magnitude as HyCC. However, they are concretely less efficient for MPC in terms non-free AND gates (by factor 1.3x to 2.1x). Especially the biomatch circuit is only half the size when compiled with HyCC. This is due to the fact that HyCC has circuit-level optimizations, making it possible to remove the highly redundant instructions coming from loop unrolling.

However, our fully automatic post-processing significantly lowers this disadvantage, making the Gauss application even more efficient than when compiled with HyCC. In terms of post-processing for post-quantum MPC, we have to compare the total number of gates (cf. Sect. 2.1). There, our post-quantum post-processing manages to reduce the size by up to factor 1.7x compared to the regular LLVM output and improves up to factor 1.3x upon the already post-processed version for MPC considering free-XOR [35].

Concrete Efficiency. In Table 4, we present the performance measurements when executing the circuits with the BMR protocol (cf. Sect. 2.1) in the recent MOTION framework [11] for up to $N = 5$ parties. The goal of this evaluation is to determine how the differences in circuit quality effect concrete performance, especially in comparison with the circuits generated by HyCC [13]. All benchmarks were performed on machines with an Intel Core i9-7960X CPU @ 2.80 GHz and 128 GB of RAM. Each party has a dedicated machine, communicating via 10 Gbit/s Ethernet with 1.25 ms RTT. Additionally, we simulate a 100 Mbit/s WAN connection with 100 ms RTT to model secure computation over the Internet.

Table 3. Circuit size of different programs compiled with HyCC and LLVM, and with post-processing for minimizing the number of AND gates (LLVM$^+$) and the total number of gates for more efficient post-quantum MPC (LLVM PQ$^+$).

	Number of gates in thousands formatted as "non-XOR/total"			
Program	HyCC [13]	LLVM	LLVM$^+$	LLVM PQ$^+$
Euclid	1.47 / 5.24	1.99 / 6.46	1.47 / 6.97	1.47 / 5.26
Dummy	1.59 / 5.30	2.19 / 6.05	1.74 / 7.40	1.74 / 5.78
Gauss	39.15 / 114.18	39.40 / 114.33	38.91 / 133.49	38.91 / 113.42
Biomatch	22.67 / 67.93	47.14 / 135.99	27.66 / 81.72	27.69 / 78.23

Table 4. Communication and runtime for running our applications using the BMR protocol [5] in the MOTION framework [11] with N parties.

Program	N	Communication [MB]			Runtime LAN [s]			Runtime WAN [s]		
		HyCC [13]	LLVM	LLVM$^+$	HyCC [13]	LLVM	LLVM$^+$	HyCC [13]	LLVM	LLVM$^+$
Euclid	2	0.88	1.17	0.88	0.18	0.22	0.19	1.14	1.20	1.17
	3	0.97	1.29	0.97	0.24	0.26	0.22	1.32	1.37	1.26
	4	1.06	1.41	1.06	0.27	0.33	0.28	1.38	1.56	1.40
	5	1.15	1.54	1.15	0.34	0.36	0.31	1.57	1.86	1.58
Dummy	2	0.95	1.29	1.03	0.19	0.23	0.20	1.13	1.26	1.20
	3	1.04	1.42	1.14	0.24	0.26	0.26	1.22	1.31	1.32
	4	1.14	1.55	1.24	0.29	0.30	0.29	1.37	1.59	1.43
	5	1.24	1.69	1.35	0.35	0.37	0.33	1.57	1.84	1.76
Gauss	2	22.45	22.59	22.31	1.79	1.92	1.80	3.97	3.98	3.90
	3	24.84	24.99	24.68	1.86	1.90	1.90	7.44	7.67	7.65
	4	27.23	27.40	27.06	2.16	2.19	2.16	12.14	11.65	10.95
	5	29.62	29.80	29.43	2.47	2.51	2.56	15.79	16.24	15.48
Biomatch	2	13.01	27.01	15.86	1.97	3.69	1.79	3.81	6.43	4.00
	3	14.39	29.88	17.55	1.94	3.73	2.04	6.20	10.32	6.30
	4	15.78	32.76	19.24	2.15	4.05	2.14	8.35	15.53	9.23
	5	17.16	35.64	20.93	2.50	4.57	2.46	11.34	19.67	12.23

As expected, the performance strongly correlates with the size of the generated circuit. We can observe that while the runtime and communication for LLVM-generated circuits is already in the same ballpark as HyCC, our post-processing diminishes the additional overhead such that our circuits perform almost equally (Euclid, dummy, biomatch), and sometimes even better (Gauss). The biggest impact of post-processing can again be seen for the biomatch application, where we are able to cut runtime and communication almost by half by removing highly redundant parts of the circuit.

7 Conclusion and Outlook

Our LLVM-based approach to MPC circuit compilation is promising, especially in terms of extensibility, usability, and circuit quality. Supporting different non-domain specific programming languages (currently C, C++, and Fortran), we make MPC practical for various software developer communities.

In the following, we give a comprehensive outlook by discussing remaining limitations regarding LLVM-IR language support, extensions to other front-ends, and the generation of hybrid circuits for mixed-protocol MPC.

Language Support. Our LLVM optimizer-based approach currently does not support structs and recursive functions. Struct support can be added as part of the gateify pass (cf. Sect. 4.3), or a dedicated struct remove pass. Partial inlining of recursive functions has been a field of interest of the LLVM community since 2015 [4] as it can increase performance of recursive programs [53]. With exception of tail call optimized recursion, however, no optimization pass has been developed until now.

Extension to Other Front-Ends. The highlight of our approach is its independence of the compiled high-level language, as we only operate on LLVM-IR, which is shared among all front-ends. Unfortunately, front-ends for different programming languages compile to vastly different LLVM-IR code. For example, a simple program that returns the addition of two integers compiles to ∼17 kB of LLVM-IR code when written in Rust, while a C version is only ∼2 kB. This is because Clang almost directly translates C code to LLVM-IR, while Rust makes heavy use of LLVM's intrinsic functions (e.g., `llvm.sadd.with.overflow.*`). In case of errors like overflow, underflow, type errors, out of bounds memory accesses, or similar, probably unwanted behavior, the code tries to recover or terminate the program with a meaningful error message. Translating all these extra steps in a circuit would lead to massive circuits.

We suggest to develop a transform pass that tries to remove most of the checks and error states. Thus, only circuit logic for essential parts of the program is generated, while keeping the program and circuit equivalent for valid inputs.

Hybrid Circuits. HyCC [13] generates Boolean and arithmetic circuits for mixed-protocol MPC. In contrast, our work only studies size-optimized Boolean circuits. A first step for achieving parity in this regard would be to equip the gateify pass with suitable building blocks (e.g., depth-optimized Boolean circuits for GMW [23]) and to allow direct translation of arithmetic LLVM-IR operations like `add`. As for finding the optimal protocol selection, we propose to implement a suitable heuristic that gathers and analyzes all relevant information during an immutable pass and divides/annotates the program in a module/analysis pass. Any of this would also require a significant extension of the Bristol format to support arithmetic operations and annotations for protocol conversions.

Acknowledgment. We thank the anonymous reviewers for their helpful comments. This project was co-funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230, and by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE. It has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI).

A Optimized Building Blocks

We provide details of the building blocks used by our LLVM toolchain during the gateify pass (cf. Sect. 4.3). To obtain these building blocks, we utilize logic synthesis tools [1, 57, 62] with our custom technology libraries (cf. Sect. 5.2) to optimize (multiplicative) size and restrict the types of basic gates. The most common building blocks are addition, subtraction, multiplication, and (integer) division, multiplexer for array accesses, and comparator, which we detail in the following. Table 5 shows a summary of the circuit size complexities, i.e., the number of non-linear (AND) gates. Moreover, we show the actual circuit sizes for standard 32 bit integers generated by the synthesis tool.

Table 5. Multiplicative complexity of building blocks for bit length l. Concrete sizes for $l = 32$ as used in Sect. 4.3.

Building block	ADD	SUB	MULT	DIV	MUX	CMP
Multiplicative complexity (# non-XOR)	$l-1$	$l-1$	l^2-l	l^2+2l+1	l	l
Concrete size ($l = 32$)	31	31	993	1264	32	32

Addition/Subtraction. To perform addition of two l-bit values, the traditional ripple carry adder (RCA), in which the carry out of one stage is fed directly to the carry-in of the next stage, has a multiplicative size of $l-1$ [10, 34]. The subtractor can be viewed as a special case of adder as the subtraction of two values a and b can be represented as $a - \bar{b} + 1$ where $\bar{b}$ denotes the two's complement representation of b.

Multiplication. In classic logic synthesis, a multiplier outputs a $2l$-bit product of two l-bit inputs. The best approach for this multiplier is the textbook method with the size of $2l^2 - l$ [34]. However, in many programming languages and MPC protocols, multiplication is defined as a $l \rightarrow l$ operation, where the product of two l unsigned integers is l-bit. Generating a $l \rightarrow l$ multiplication with logic synthesis tools give us a circuit size of $l^2 - l$ [27, 44].

Division. The division operation computes the quotient and remainder of two binary integer numbers. The standard approach for the division is similar to the text-book multiplication, where the divisor is iteratively shifted and subtracted from the remainder. By doing so, one division operation can be built with complexity of $2l^2$ AND gates. Restoring division can help us in hardware synthesis to have a complexity of $l^2 + 2l + 1$ [51].

Multiplexer. A 2-to-1 MUX was proposed in [35] with a size of l. The tree architecture for an m-to-1 MUX has size $(m-1)l$.

Comparator. The standard comparator circuit checks whether one l-bit number is greater than another with a size of l. We implement this comparator as described in [35].

B Dummy Application

In Listing 1 we provide the C++ code for our dummy application that we use for benchmark purposes in addition to applications from the HyCC repository (cf. Sect. 6.2). It showcases as many supported features as short as possible.

Listing 1. Dummy application that covers many supported features.

```
#include <stdio.h>

int dummy (int a, int b, int c) {
    int array[8];
    for (int i=0; i<8; i++) {
        array[i] = a + b * i;
    }
    int ret=0;
    if (c < array[c]) {
        ret = array[2] + array[3];
    }
    else {
        ret = array[0] * array[1];
    }
    return ret;
}

int main(){
    int a, b, c;
    scanf("%d\n%d\n%d", &a, &b, &c);
    printf("DummyFunction: %d\n", dummy(a, b, c));
}
```

References

1. ABC: A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~alanmi/abc/
2. Aly, A., et al.: SCALE-MAMBA v1. 10: Documentation (2020)
3. Archer, D., et al.: Bristol Fashion MPC circuits (2020). https://homes.esat.kuleuven.be/~nsmart/MPC/
4. Barrio, P., Carruth, C., Molloy, J.: Recursion inlining in LLVM (2015). https://www.llvm.org/devmtg/2015-04/slides/recursion-inlining-2015.pdf
5. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols (extended abstract). In: STOC (1990)
6. Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: CCS (2008)
7. Boemer, F., Cammarota, R., Demmler, D., Schneider, T., Yalame, H.: MP2ML: a mixed-protocol machine learning framework for private inference. In: ARES (2020)
8. 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). https://doi.org/10.1007/978-3-540-88313-5_13

9. Boston University: JIFF: JavaScript implementation of federated functionalities (2015). https://github.com/multiparty/jiff/
10. Boyar, J., Damgård, I., Peralta, R.: Short non-interactive cryptographic proofs. J. Cryptol. **13**, 449–472 (2000)
11. Braun, L., Demmler, D., Schneider, T., Tkachenko, O.: MOTION - A framework for mixed-protocol multi-party computation. ePrint (2020). https://ia.cr/2020/1137
12. Büscher, N., et al.: Secure two-party computation in a quantum world. In: Conti, M., Zhou, J., Casalicchio, E., Spognardi, A. (eds.) ACNS 2020. LNCS, vol. 12146, pp. 461–480. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57808-4_23
13. Büscher, N., Demmler, D., Katzenbeisser, S., Kretzmer, D., Schneider, T.: HyCC: compilation of hybrid protocols for practical secure computation. In: CCS (2018)
14. Chandran, N., Gupta, D., Rastogi, A., Sharma, R., Tripathi, S.: EzPC: programmable, efficient, and scalable secure two-party computation for machine learning. In: EuroS&P (2019)
15. Clarke, E., Kroening, 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). https://doi.org/10.1007/978-3-540-24730-2_15
16. Coussy, P., Morawiec, A.: High-Level Synthesis: From Algorithm to Digital Circuit. Springer, Dordrecht (2008). https://doi.org/10.1007/978-1-4020-8588-8
17. Demmler, D., Dessouky, G., Koushanfar, F., Sadeghi, A., Schneider, T., Zeitouni, S.: Automated synthesis of optimized circuits for secure computation. In: CCS (2015)
18. Demmler, D., Schneider, T., Zohner, M.: ABY - a framework for efficient mixed-protocol secure two-party computation. In: NDSS (2015)
19. Dessouky, G., Koushanfar, F., Sadeghi, A.R., Schneider, T., Zeitouni, S., Zohner, M.: Pushing the communication barrier in secure computation using lookup tables. In: NDSS (2017)
20. Fereidooni, H., et al.: SAFELearn: secure aggregation for private federated learning. In: Deep Learning and Security Workshop (2021)
21. Fowler, D., Robson, E.: Square root approximations in old Babylonian mathematics: YBC 7289 in context. Historia Mathematica (1998)
22. Fraser, C.W., Hanson, D.R.: A Retargetable C Compiler: Design and Implementation. Addison-Wesley (1995)
23. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC (1987)
24. Gueron, S., Lindell, Y., Nof, A., Pinkas, B.: Fast garbling of circuits under standard assumptions. In: CCS, pp. 567–578. ACM (2015)
25. Hastings, M., Hemenway, B., Noble, D., Zdancewic, S.: SoK: general purpose compilers for secure multi-party computation. In: S&P (2019)
26. Henecka, W., Kögl, S., Sadeghi, A., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: CCS (2010)
27. Holzer, A., Franz, M., Katzenbeisser, S., Veith, H.: Secure two-party computations in ANSI C. In: CCS (2012)
28. Huang, Y., Evans, D., Katz, J.: Private set intersection: are garbled circuits better than custom protocols? In: NDSS (2012)
29. Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: USENIX Security (2011)
30. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 145–161. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_9

31. Ishaq, M., Milanova, A.L., Zikas, V.: Efficient MPC via program analysis: a framework for efficient optimal mixing. In: CCS (2019)
32. Javadi, M., Yalame, H., Mahdiani, H.: Small constant mean-error imprecise adder/-multiplier for efficient VLSI implementation of MAC-based applications. IEEE Trans. Comput. (2020)
33. Keller, M.: MP-SPDZ: a versatile framework for multi-party computation. In: CCS (2020)
34. 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). https://doi.org/10.1007/978-3-642-10433-6_1
35. 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. LNCS, vol. 5126, pp. 486–498. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70583-3_40
36. Kreuter, B., Shelat, A., Mood, B., Butler, K.: PCF: a portable circuit format for scalable two-party secure computation. In: USENIX Security (2013)
37. Kreuter, B., Shelat, A., Shen, C.H.: Billion-gate secure computation with malicious adversaries. In: USENIX Security (2012)
38. Lattner, C., Adve, V.S.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Code Generation and Optimization (2004)
39. LLVM Community: CIRCT / Circuit IR compilers and tools (2020). https://github.com/llvm/circt
40. LLVM Project: LLVM's analysis and transform passes (2020). https://llvm.org/docs/Passes.html
41. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay - Secure two-party computation system. In: USENIX Security (2004)
42. Mantel, H., Scheidel, L., Schneider, T., Weber, A., Weinert, C., Weißmantel, T.: RiCaSi: rigorous cache side channel mitigation via selective circuit compilation. In: Krenn, S., Shulman, H., Vaudenay, S. (eds.) CANS 2020. LNCS, vol. 12579, pp. 505–525. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-65411-5_25
43. Mohassel, P., Rindal, P.: ABY3: a mixed protocol framework for machine learning. In: CCS (2018)
44. Mood, B., Gupta, D., Carter, H., Butler, K.R.B., Traynor, P.: Frigate: a validated, extensible, and efficient compiler and interpreter for secure computation. In: Euro S&P (2016)
45. 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). https://doi.org/10.1007/978-3-642-32946-3_19
46. Nguyen, T.D., et al.: FLGUARD: secure and private federated learning. ePrint (2021). https://ia.cr/2021/025
47. Nielsen, J.D., Schwartzbach, M.I.: A domain-specific programming language for secure multiparty computation. In: Workshop on Programming Languages and Analysis for Security (2007)
48. Patra, A., Schneider, T., Suresh, A., Yalame, H.: ABY2.0: improved mixed-protocol secure two-party computation. In: USENIX Security (2020)
49. Pinkas, B., Rosulek, M., Trieu, N., Yanai, A.: PSI from PaXoS: fast, malicious private set intersection. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12106, pp. 739–767. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45724-2_25

50. Rastogi, A., Hammer, M.A., Hicks, M.: Wysteria: a programming language for generic, mixed-mode multiparty computations. In: S&P (2014)
51. Robertson, J.E.: A new class of digital division methods. Trans. Electron. Comput. (1958)
52. Rotaru, D.: awesome-mpc (2020). https://github.com/rdragos/awesome-mpc#frameworks
53. Rugina, R., Rinard, M.: Recursion unrolling for divide and conquer programs. In: Midkiff, S.P., et al. (eds.) LCPC 2000. LNCS, vol. 2017, pp. 34–48. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45574-4_3
54. Schoenmakers, B.: MPyC: secure multiparty computation in Python (2018). https://github.com/lschoe/mpyc/blob/master/README.md
55. Schropfer, A., Kerschbaum, F., Muller, G.: L1 - an intermediate language for mixed-protocol secure computation. In: Computer Software and Applications Conference (2011)
56. Songhori, E.M., Hussain, S.U., Sadeghi, A., Schneider, T., Koushanfar, F.: TinyGarble: highly compressed and scalable sequential garbled circuits. In: S&P (2015)
57. Synopsis: DC Ultra (2020). https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html
58. Tatsuoka, M., et al.: Physically aware high level synthesis design flow. In: DAC (2015)
59. Tillich, S., Smart, N.: (Bristol Format) Circuits of basic functions suitable for MPC and FHE (2020). https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html
60. Verilog.com: Verilog Resources (2020). https://verilog.com/
61. Wang, X., Malozemoff, A.J., Katz, J.: EMP-toolkit: Efficient multiparty computation toolkit (2016). https://github.com/emp-toolkit
62. Wolf, C.: Yosys open synthesis suite. http://www.clifford.at/yosys/
63. Yao, A.C.: How to generate and exchange secrets (extended abstract). In: FOCS (1986)
64. Zahur, S., Evans, D.: Obliv-C: a language for extensible data-oblivious computation. ePrint (2015). https://ia.cr/2015/1153
65. Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 220–250. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_8
66. Zhang, Y., Steele, A., Blanton, M.: PICCO: A general-purpose compiler for private distributed computation. In: CCS (2013)