# Improved Circuit Compilation for Hybrid MPC via Compiler Intermediate Representation

Daniel Demmler[1], Stefan Katzenbeisser[2],
Thomas Schneider[3], Tom Schuster[3] and Christian Weinert[3]
[1]*University of Hamburg, Germany*
[2]*University of Passau, Germany*
[3]*Technical University of Darmstadt, Germany*

Keywords: Privacy-preserving Protocols, MPC, Circuit Compilation.

Abstract: Secure multi-party computation (MPC) allows multiple parties to securely evaluate a public function on their private inputs. The field has steadily moved forward and real-world applications have become practical. However, MPC implementations are often hand-built and require cryptographic knowledge. Thus, special compilers like HyCC (Büscher et al., CCS'18) have been developed, which automatically compile high-level programs to combinations of Boolean and arithmetic circuits required for mixed-protocol (hybrid) MPC. In this work, we explore the advantages of extending MPC compilers with an intermediate representation (IR) as commonly used in modern compiler infrastructures. For this, we extend HyCC with a graph-based IR that facilitates the implementation of well-known algorithms from compiler design as well as further MPC-specific optimizations. We demonstrate the benefits by implementing arithmetic decomposition based on our new IR that automatically extracts arithmetic expressions and then compiles them into separate circuits. For a line intersection algorithm, we require 40 % less run-time and improve total communication by a factor of 3× compared to regular HyCC when securely evaluating the corresponding circuit with the hybrid MPC framework ABY (Demmler et al., NDSS'15).



## 1 INTRODUCTION

Secure multi-party computation (MPC) allows two or more mutually distrusting parties to securely evaluate an arbitrary public function on their private inputs such that only the result of the computation is revealed but nothing else. MPC was already introduced in the 1980s with Yao's garbled circuits (Yao, 1986) and the protocol by Goldreich, Micali, and Wigderson (GMW) (Goldreich et al., 1987).

Only in recent years, research has lead to efficient MPC implementations that can make applications practical in the real world, e.g., private tax fraud detection in Estonia (Bogdanov et al., 2015) and measuring ad conversion rates at Google (Ion et al., 2020). Also, companies like Sharemind or Unbound Tech emerge that provide MPC-based software solutions as alternatives to trusted hardware (Archer et al., 2018).

So far, however, extensive cryptographic knowledge was required to implement applications in MPC, where functions must be expressed as Boolean or arithmetic circuits, or a combination thereof. Manual effort was especially necessary to utilize *hybrid* protocols, which can greatly improve the overall performance by *mixing* multiple MPC protocols to evaluate a complex function efficiently (Demmler et al., 2015b).

Specialized compilers like HyCC (Büscher et al., 2018) made significant steps towards the broader use of MPC by automatically translating high-level programming languages like ANSI C into combinations of Boolean and arithmetic circuits, which are common representations of functions in MPC. Specifically, circuits for MPC represent functions as a set of primitive gates (e.g., *AND*, *XOR*, *NOT*, *ADD*, or *MULT*) interconnected via wires, similar to digital logic circuits.

However, MPC compilers lack many optimizations that are commonly implemented in compiler infrastructures such as LLVM (Lattner and Adve, 2004), which mostly rely on the internal use of a graph-based *intermediate representation* (IR) (Aho et al., 1986).

**Our Contributions.** In this work, we show that MPC compilers can be extended with an intermediate repre-



Demmler, D., Katzenbeisser, S., Schneider, T., Schuster, T. and Weinert, C.
Improved Circuit Compilation for Hybrid MPC via Compiler Intermediate Representation.
DOI: 10.5220/0010540504440451
In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 444-451
ISBN: 978-989-758-524-1

sentation similar to modern compiler infrastructures, resulting in significant enhancements for circuit compilation. We enable new optimizations and improve the size as well as composition of compiled circuits, thus increasing the performance when evaluating these circuits with MPC.

For this, we design a new graph-based IR inspired by LLVM (Lattner and Adve, 2004) and integrate it into the state-of-the-art MPC compiler HyCC (Büscher et al., 2018) as a new pipeline phase. Our IR is especially designed to facilitate the implementation of well-known algorithms from compiler design as well as further MPC-specific optimizations.

In order to demonstrate its benefits, we design and implement a new compiler optimization for *arithmetic decomposition* based on our new IR. The MPC compiler can now automatically extract arithmetic operations from larger units of code to allow these operations to be evaluated using an MPC protocol that is especially suitable for additions and multiplications. Arithmetic decomposition was already discussed theoretically in (Büscher et al., 2018), but only a developer-guided, coarse-grained version based on function decomposition has been implemented and evaluated. In this paper, we give a completely automated solution.

We also demonstrate the improvements that our approach achieves by measuring the performance for evaluating circuits generated by HyCC in the two-party hybrid-protocol MPC framework ABY (Demmler et al., 2015b) with and without our new optimizations. For a line intersection algorithm, where arithmetic expressions are mixed with logical operations and division, we find that arithmetic decomposition paired with an appropriate protocol selection improves the total run-time by 40 %. Moreover, the communication overhead is reduced by a factor of 3×.

We summarize our contributions as follows:

- We design a graph-based IR and integrate it into HyCC (Büscher et al., 2018).
- Using our new IR, we implement an arithmetic decomposition algorithm.
- We empirically measure our optimization by evaluating circuits for a line intersection algorithm in ABY (Demmler et al., 2015b). We observe improvements in communication of up to factor 3×.

# 2 RELATED WORK

In recent years, many MPC compilers and frameworks have been proposed, which are extensively reviewed in (Hastings et al., 2019). TASTY (Henecka et al., 2010) and ABY (Demmler et al., 2015b) demonstrated that hybrid secure protocols can significantly increase performance. However, their benchmarks were hand-built, whereas we focus on automatically generating highly efficient hybrid circuits.

TinyGarble (Songhori et al., 2015) and (Demmler et al., 2015a; Testa et al., 2019) created optimized circuits from hardware description languages, which, however, are not widely known among regular software developers. Other works thus studied compilation from high-level languages, e.g., EzPC (Chandran et al., 2019) compiles a domain-specific language to a mix of Yao's garbled circuits and arithmetic GMW.

Our work directly extends the HyCC compiler (Büscher et al., 2018), which generates hybrid circuits optimized for evaluation with three different MPC protocols from ANSI C code. Originally, HyCC achieves a developer-guided and coarse-grained code decomposition solely based on functions and loops in the program to be compiled. We improve the decomposition by automatically detecting and extracting blocks of arithmetic statements (cf. §4.2) for a fine-grained decomposition and show that this leads to significant performance improvements (cf. §5).

The authors of the hybrid MPC compiler in (Ishaq et al., 2019) use static single assignment (SSA) form translated from Java for optimal protocol selection. They apply linear programming to find the most efficient assignment, which is limited to *two* MPC protocols. In contrast, we automatically optimize and compile ANSI C for three MPC protocols.

# 3 PRELIMINARIES

## 3.1 MPC Tools

We shortly introduce the MPC framework ABY that we utilize for benchmarking and the MPC compiler HyCC that we extend in our work.

### 3.1.1 ABY

The ABY framework (Demmler et al., 2015b) is a state-of-the art framework for manually mixing different two-party MPC protocols and their state-of-the-art optimizations: Yao's garbled circuits (Yao, 1986), the GMW protocol (Goldreich et al., 1987), and an arithmetic version of the GMW protocol. ABY is a hybrid-protocol framework: it is possible to securely switch between these protocols for different parts of the function to be evaluated, which often results in better overall efficiency.

#### 3.1.2 HyCC

The open-source HyCC compiler (Büscher et al., 2018) extends the CMBC-GC compiler (Holzer et al., 2012), which in turn is based on the bounded model checker CBMC (Clarke et al., 2004). HyCC automatically compiles ANSI C programs to circuits optimized for *hybrid* MPC.

The originial compiler pipeline of HyCC without our IR worked as follows (Büscher et al., 2018):

(a) *CBMC Front-end*: preprocessing, lexing, and parsing of ANSI C code resulting in goto-code.

(b) *Optimization*: static analysis, constant propagation, symbolic execution for loop unrolling.

(c) *Function Decomposition*: outlining of functions into dedicated "modules".

(d) *Circuit Compilation*: generation of size- and depth-optimized Boolean as well as arithmetic circuits for each module.

(e) *Circuit Export*: circuit export into an ABY-compatible format (Demmler et al., 2015b).

(f) *Protocol Selection*: picking the best MPC protocol for each module.

### 3.2 Compiler Background

We shortly introduce the terminology of compiler literature and common forms of compiler intermediate representations.

#### 3.2.1 Control Flow Graph

The control flow graph (CFG) of a program is a directed graph of all possible execution paths. Popular compiler optimizations like dependency analyses are usually done on CFGs. The nodes represent *basic blocks* and the edges represent jumps or transitions between them. A basic block contains one or more instructions that are executed sequentially.

A common way of iterating through CFGs is in reverse post-order (RPO): a block is visited before all of its successor blocks, unless the successor is reachable via a back edge.

#### 3.2.2 Compiler IR

In the classic compiler model (Aho et al., 1986), a front-end parses high-level source code from which first an intermediate representation (IR) is created, and a back-end subsequently produces (machine) code. The main purpose of an IR is to simplify the optimizations and analyses of a program, since human readable source code is not suitable for automated processing.

The most common forms of IR include graph-, tuple-, and stack-based IRs, which can be combined. Our graph-based IR is using flat tuple-based instructions inside its nodes. Our IR is almost in SSA form, as there are no IR-level variables. An extension to proper SSA form is therefore easily possible but not necessary for our work.

#### 3.2.3 Goto-code

The CBMC front-end represents parsed and processed ANSI C code as linear goto-code, which does not contain higher-level constructs such as `if` or `for` statements. Instead, the control flow is converted to `goto` instructions, which are similar to the `jmp` instruction and its conditional variants in x86 assembly. Goto-code has distinct concepts of instructions (e.g., `assign` or `goto`) and expressions (e.g., addition), which can be nested. A function is represented as a list of instructions that may reference expressions.

## 4 A NEW COMPILER IR FOR CIRCUIT COMPILATION

Our goal of designing and integrating a new compiler IR in HyCC is to facilitate the implementation of (possibly MPC-specific) compiler optimizations and analyses in order to improve compilation results and increase MPC performance.

HyCC already contains several data structures that can be categorized as IR, especially the goto-code representation obtained by parsing ANSI C code in the CBMC front-end. Unfortunately, goto-code is not well-suited for building compiler optimizations: It is only "stringly" typed, encodes information about variables in the name, and uses a different expression sub-class for each operation. Moreover, the return value of a function call is referenced by a magic variable name, making it error-prone to insert or remove function calls. Furthermore, memory access is very abstract, rendering data-dependency analyses impractical. Finally, iterating through all instructions in a function in reverse post-order (RPO) is cumbersome, but required for many data-flow analyses.

Our new compiler IR, simply called "IR" in the following, remains relatively high-level, but abstracts away a lot of the unnecessary details that goto-code contains. It also uses the concept of basic blocks, but instead of using goto instructions to jump to labeled instructions inside the linear code representing the program, our new IR uses a graph-based representation: Every basic block has a pointer to the head of the

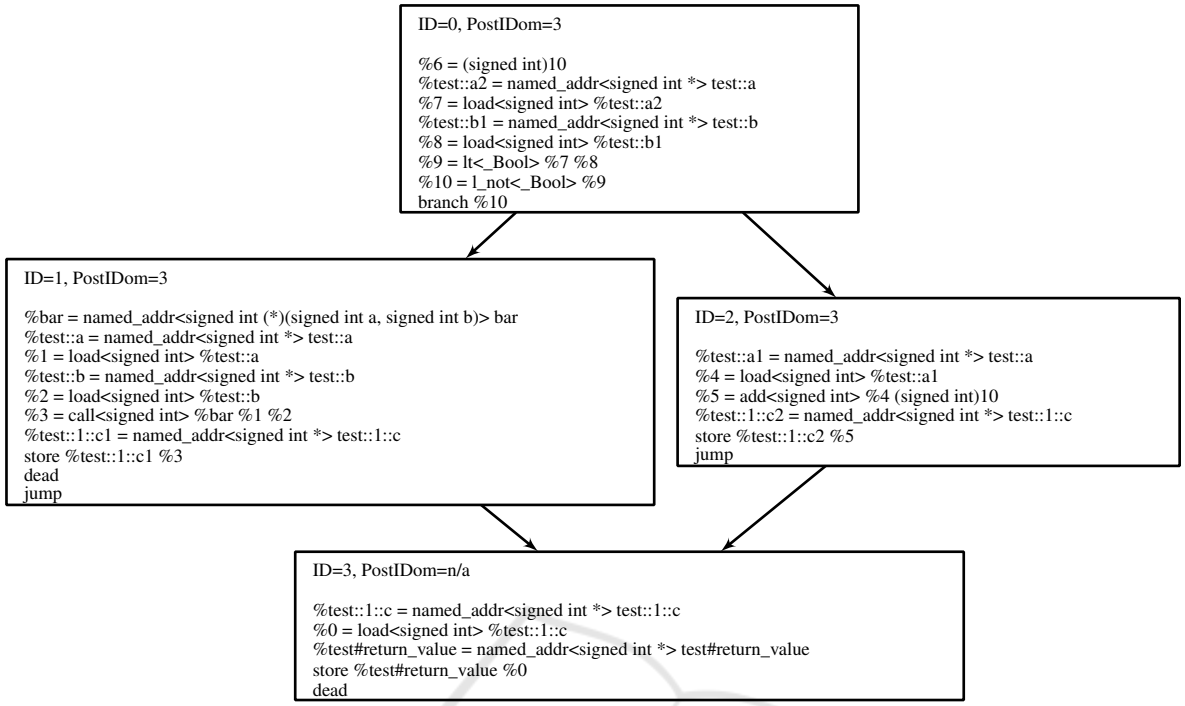


Figure 1: Graph-based representation of the code from Lst. 1 in our new IR.

linked list of instructions. We also maintain a set of incoming and outgoing edges for every block.

Our IR is a new stage in the HyCC compiler pipeline described in §3.1 and located between the CBMC front-end (a) and the optimization phase (b). Its design is inspired by modern compiler infrastructures, especially LLVM (Lattner and Adve, 2004). It consists of only one instruction class `Instr`, compared to the separation of expressions and instructions in goto-code. The new simplistic instructions and the CFG with basic blocks enable us to implement common analysis passes as described in the compiler literature. We briefly describe all `Instr` fields:

**Type.** The C data type associated with expressions (e.g., `signed int`), which can be copied directly from goto-code. Almost every instruction has a type, with the notable exception of jump and branch instructions.

Listing 1: C code representation of test function.

```
1  int test(int a, int b) {
2      int c;
3      if (a < b) {
4          c = a + 10;
5      } else {
6          c = bar(a, b);
7      }
8      return c;
9  }
```

**Kind.** The kind of this instruction, e.g., addition, constant, call, or store.

**Block.** The basic block containing this instruction.

**Operands.** For an add instruction, these are two instructions representing both sides of the expression. Instructions such as `call` have one operand for each argument.

This field is a pointer to another instruction previously contained in the linked list. Unlike expressions in goto-code, there is no direct nesting.

**Next.** Pointer to the next instruction in the basic block.

We give an example in Fig. 1. This graph represents our IR for the C code shown in Lst. 1. There are four nodes, i.e., basic blocks. The four edges show the possible control flow between the blocks. Every block also has a unique ID. *PostIDom* is the ID of the immediate post dominator of that block, i.e., the nearest block that is guaranteed to be executed at some point after this block. The *entry block* of the graph has ID 0 and contains the condition for the if-statement. The if- and else-blocks have IDs 2 and 3, respectively. The *exit block* has ID 3.

A major difference between our IR and goto-code is the introduction of explicit `load` and `store` instructions. In IR, we only refer to the address of a variable with `named_addr`. In C, this equals creating a reference with the ampersand operator, i.e., `&a`. Loading the value of a variable then involves a `load`

of the named_addr. By introducing these additional instructions, we make the IR easier to analyze: we can ensure that every memory access happens only using load and store instructions. This becomes more interesting when considering the many other forms of memory accesses in C, which all use different expressions in goto-code. For example, struct members (e.g., position.x in C and member_expr in goto-code), array access (e.g., array[i] in C and index_expr in goto-code), and manual pointer arithmetic via arithmetic expressions. All of these are unified in our IR using a single instruction: compute_addr.

The compute_addr instruction has a base pointer, an offset from that pointer, and a scaled-index operand. The offset parameter can be used to address a specific struct member using its memory offset. For array accesses, the scaling parameter is used to scale the index using the array's native type size to the appropriate full address. In C pointer arithmetic, this is equivalent to $\mathsf{base} + \mathsf{offset} + \mathsf{index} * \mathsf{scaling}$. This instruction is comparable to LLVM's getelementptr.

## 4.1 HyCC Integration

In the following, we describe the integration of our new graph-based IR into HyCC (Büscher et al., 2018). For this, we present conversions from/to semantically equivalent goto-code.

### 4.1.1 Goto-code to IR

The conversion from flat goto-code to our graph-based IR first generates basic blocks: For each goto jump target, a new basic block is created, unless it already exists. Additionally, a goto also marks the end of the current basic block and a new basic block is created for the next instruction.

GOTO instructions are converted to jump or branch instructions, if they are conditional. For every ASSIGN instruction, a store instruction is created; FUNCTIONAL_CALL instructions result in call instructions. For goto expressions like addition, there exists a one-to-one mapping to a new Instr instance of that specific kind. References to variables (symbol_exprt expressions) are changed to named_addr. More complex instructions like array indexing or struct member access produce the new compute_addr kind.

### 4.1.2 IR to Goto-code

The remainder of the HyCC compiler pipeline only operates on goto-code. Thus, we design and implement a conversion back to goto-code, that happens after optimization and mark/extract (cf. §4.2).

The conversion algorithm mainly flattens the graph structure to linear code and separates IR instructions into goto instructions and nested expressions. Importantly, we produce goto-code that is semantically equivalent to the goto-code before the IR conversion.

The final goto-code is produced in two phases: First, all instructions are emitted. Then, the jump targets of the emitted (conditional) goto instructions are initialized. We iterate through all blocks in reverse post-order (RPO).

After emitting all goto-code, there is still bookkeeping work to be done. When decomposing a function in our IR, new structs, parameters, and variable names are introduced that the CBMC front-end is not aware of. Therefore, new symbols representing variable names and types are added to the existing symbol table(s).

## 4.2 Arithmetic Decomposition

Using our new IR, we design and implement fine-grained arithmetic decomposition by moving arithmetic expressions to separate functions created in the IR. Thereby, we take advantage of the already existing *function decomposition* in HyCC, which compiles the new functions as a new module to arithmetic circuits (cf. §3.1).

Our algorithm is basic block local. It thus operates on one basic block at a time instead of decomposing code across multiple basic blocks or even a whole function. It consists of two phases: the *mark* and the *extract* phase described next.

### 4.2.1 Mark Phase

Iterating through all instructions inside a basic block, we only inspect those with an observable side effect, i.e., call, store, and jump/branch. The main goal is to group as many adjacent store instructions as possible.

For every store instruction, we make sure it is *extractable*, i.e., it can be moved to a new IR-level function dedicated to arithmetic operations. For this, we check three conditions: First, we ensure that the *left-hand-side* (lhs) of the assignment is referencing a regular variable. Second, trivial expressions like constants or variable references are skipped. Finally, a heuristic is applied to the last store instruction inside a basic block to avoid extraction of loop counter operations.

For the remaining instructions, we recursively iterate through the expressions on the *right-hand side* (rhs) and ensure the operands are supported by arithmetic circuits, whereas all types of logic and bitwise operations are implicitly rejected. The full mark algorithm is given in Alg. 1.

Algorithm 1: The mark algorithm.

```
procedure EXTRACTABLE(expr)
    if expr.kind ∈ {addition, subtraction,
                    multiplication} then
        return EXTRACTABLE(expr.lhs) and
               EXTRACTABLE(expr.rhs)
    end if
    if expr.kind = load then
        return EXTRACTABLE(expr.operand)
    end if
    if expr.kind ∈ {constant, named_addr} then
        return true
    end if
    return false
end procedure
```

Algorithm 2: The extract algorithm.

```
procedure EXTRACT(instr, is_lhs)
    if instr.kind = named_addr then
        name ← instr.variable_name
        if is_lhs then
            assignment ← assignment + name
        else if name ∉ assignment then
            ADD_PARAMETER(name)
        end if
        name ← RENAME(name)
        return NEW_NAMED_ADDR(name)
    end if
    if instr.kind = store then
        rhs ← EXTRACT(instr.rhs, false)
        lhs ← EXTRACT(instr.lhs, true)
        return NEW_STORE(lhs, rhs)
    end if
    for all op ∈ instr.ops do
        op ← EXTRACT(op, is_lhs)
    end for
    return COPY(instr, instr.ops)
end procedure
```

#### 4.2.2 Extract Phase

If a store instruction is the first instruction in a group of marked store instructions, a new IR-level function is created. All subsequent store instructions and their reachable operands are moved to that new function.

While most instructions like addition and multiplication can simply be copied to the new function, special care has to be taken regarding references to named variables. The variable names were created in the context of the original function and encode various information that do not apply to the new function.

Copying instructions to a new function and appropriately renaming variables is not enough. We furthermore need to add parameters for references to variables that were not moved to the new function. In more complex scenarios, there are also references to variables in outer blocks. Of course, we also need to add return values for all the variables that were assigned by the extracted store instructions.

Finally, the original function must be updated. All extracted store instructions are removed and replaced with a single call instruction to the newly created function. All required variables are added as arguments and the return values are assigned appropriately.

As a result, after conversion back to goto-code, the decomposition phase (c) in the HyCC pipeline (cf. §3.1) can outline each new function into a dedicated module, which subsequently can be successfully compiled to an arithmetic circuit (d), and eventually be evaluated with an appropriate arithmetic protocol (f). The full extract algorithm is given in Alg. 2.

## 5 EVALUATION

We evaluate our extension of HyCC with our new compiler IR and the arithmetic decomposition optimization using a line intersection algorithm from rosettacode.org (Rosetta Code, 2020) where arithmetic expressions are mixed with logical operations and division.

Finding the intersection of two lines in Euclidean geometry has interesting practical use cases, e.g., in computer graphics, finance, motion planning, and collision detection. We only modify the code slightly to replace floating-point numbers with integers and inline the function for calculating the determinant to challenge our fine-grained arithmetic decomposition, as shown in Lst. 2 in the appendix.

We now compare the circuits that are generated with and without arithmetic decomposition. Without decomposition, the C program is compiled into a single Boolean circuit. With arithmetic decomposition enabled, an additional arithmetic circuit is generated.

As shown in Tab. 1, arithmetic decomposition decreases the total number of gates and notably also the number of AND gates in the hybrid circuit by more than 80 %. Since AND gates require computation and communication in most MPC protocols, we show that our optimization is well-suited for its intended usage in MPC compilers. The resulting 10 MUL and 9 SUB gates exactly match the number of multiplications and subtractions in the intersection function.

We evaluate the performance of the automatically generated circuits in ABY (Demmler et al., 2015b) for obtaining empirical performance results. Our setup consists of two machines with Intel Core i7-4790 CPUs at 3.6 GHz and 32 GB RAM connected in a 10 Gbit/s LAN with 1 ms RTT.

The measured run-times (averaged over 10 executions) and communication results are summarized in Tab. 2. We split the measurements into an input-independent *setup phase* that can be pre-computed and an *online phase* that is executed once the private

Table 1: Number and types of gates with and without arithmetic decomposition.

| | AND | OR | XOR | NOT | MUL | ADD | SUB | NEG | Total |
|---|---|---|---|---|---|---|---|---|---|
| No arith. decomp. (HyCC (Büscher et al., 2018)) | 13,995 | 127 | 24,436 | 644 | 0 | 0 | 0 | 0 | 39,202 |
| With arith. decomp. (this work) | **2,307** | 211 | **4,260** | **300** | **10** | 0 | **9** | 0 | **7,097** |

Table 2: Run-times and communication for line intersection (cf. §6) in ABY (Demmler et al., 2015b).

| | Run-time [ms] | | | Communication [kB] | | |
|---|---|---|---|---|---|---|
| Protocol | Setup | Online | Total | Setup | Online | Total |
| Yao's GC (HyCC (Büscher et al., 2018)) | 5.348 | 3.515 | 8.863 | 399.79 | 6.08 | 405.86 |
| Yao's GC + Arithmetic (this work) | **3.468** | **2.081** | **5.550** | **116.07** | 19.48 | **135.56** |
| GMW (HyCC (Büscher et al., 2018)) | 6.634 | 46.095 | 52.729 | 600.10 | 19.60 | 619.71 |
| GMW + Arithmetic (this work) | **4.779** | **37.454** | **42.233** | **277.47** | 33.09 | **310.55** |

inputs are known. Boolean circuits are evaluated either with Yao's gabled circuits (GC) (Yao, 1986) or the GMW protocol (Goldreich et al., 1987) as implemented in ABY (Demmler et al., 2015b).

We observe that the total run-time is significantly improved due to arithmetic decomposition, e.g., by 40 % from 8.9 ms to 5.6 ms for the favorable combination with Yao's GC. Moreover, we improve the total communication by factor 3×. Even for the less efficient combination with GMW, the improvements are significant, e.g., the total communication is improved by factor 2×.

# 6 CONCLUSION

In this work, we designed and integrated a new graph-based compiler intermediate representation for the state-of-the-art hybrid MPC compiler HyCC (Büscher et al., 2018). We furthermore successfully utilized and evaluated our new IR for implementing arithmetic decomposition, thereby significantly improving the MPC performance when compiling, e.g., scientific or statistical applications with mixed Boolean and arithmetic operations. In the following, we shortly outline interesting areas for future work to unlock further potential of our new IR and arithmetic decomposition in HyCC.

Our new IR can be used for easily implementing more optimizations that potentially lead to further performance improvements in MPC, e.g., field-sensitive program dependence analysis (Litvak et al., 2010). For arithmetic decomposition, so far only instructions are marked as extractable if the right-hand-side of the assignment is fully decomposable. However, an improved version could also extract arithmetic subexpressions of bigger expressions with mixed operations. Moreover, C programmers often use shift operators instead of multiplication or division, because of better performance on general-purpose computers. In light of compilation for MPC protocols, left shifts by a constant should be replaced with an arithmetic multiplication with a power of two.

# ACKNOWLEDGEMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 850990 PSOTI). It was supported by the DFG as part of project E4 within the CRC 1119 CROSSING and project A.1 within the RTG 2050 "Privacy and Trust for Mobile Users", and by the BMBF and HMWK within ATHENE.

# REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, techniques, and tools*. Addison-Wesley series in computer science. Addison-Wesley.

Archer, D. W., Bogdanov, D., Lindell, Y., Kamm, L., Nielsen, K., Pagter, J. I., Smart, N. P., and Wright, R. N. (2018). From keys to databases - real-world applications of secure multi-party computation. *The Computer Journal*.

Bogdanov, D., Jõemets, M., Siim, S., and Vaht, M. (2015). How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *FC*. Springer.

Büscher, N., Demmler, D., Katzenbeisser, S., Kretzmer, D., and Schneider, T. (2018). HyCC: Compilation of hybrid protocols for practical secure computation. In *CCS*. ACM.

Chandran, N., Gupta, D., Rastogi, A., Sharma, R., and Tripathi, S. (2019). EzPC: Programmable and efficient secure two-party computation for machine learning. In *EuroS&P*. IEEE.

Clarke, E. M., Kroening, D., and Lerda, F. (2004). A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer.

Demmler, D., Dessouky, G., Koushanfar, F., Sadeghi, A., Schneider, T., and Zeitouni, S. (2015a). Automated synthesis of optimized circuits for secure computation. In *CCS*. ACM.

Demmler, D., Schneider, T., and Zohner, M. (2015b). ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*. The Internet Society.

Goldreich, O., Micali, S., and Wigderson, A. (1987). How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*. ACM.

Hastings, M., Hemenway, B., Noble, D., and Zdancewic, S. (2019). SoK: General purpose compilers for secure multi-party computation. In *S&P*. IEEE.

Henecka, W., Kögl, S., Sadeghi, A., Schneider, T., and Wehrenberg, I. (2010). TASTY: Tool for automating secure two-party computations. In *CCS*. ACM.

Holzer, A., Franz, M., Katzenbeisser, S., and Veith, H. (2012). Secure two-party computations in ANSI C. In *CCS*. ACM.

Ion, M., Kreuter, B., Nergiz, A. E., Patel, S., Saxena, S., Seth, K., Raykova, M., Shanahan, D., and Yung, M. (2020). On deploying secure computing: Private intersection-sum-with-cardinality. In *EuroS&P*. IEEE.

Ishaq, M., Milanova, A. L., and Zikas, V. (2019). Efficient MPC via program analysis: A framework for efficient optimal mixing. In *CCS*. ACM.

Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*. IEEE.

Litvak, S., Dor, N., Bodík, R., Rinetzky, N., and Sagiv, M. (2010). Field-sensitive program dependence analysis. In *FSE*. ACM.

Rosetta Code (2020). Find the intersection of two lines. https://rosettacode.org/wiki/Find_the_intersection_of_two_lines#C.2B.2B.

Songhori, E. M., Hussain, S. U., Sadeghi, A., Schneider, T., and Koushanfar, F. (2015). TinyGarble: Highly compressed and scalable sequential garbled circuits. In *S&P*. IEEE.

Testa, E., Soeken, M., Amarù, L. G., and Micheli, G. D. (2019). Reducing the multiplicative complexity in logic networks for cryptography and security applications. In *Design Automation Conference (DAC)*. ACM.

Yao, A. C. (1986). How to generate and exchange secrets. In *FOCS*. IEEE.

# APPENDIX

The line intersection algorithm code used in our evaluation (cf. §5) is based on rosettacode.org (Rosetta Code, 2020). Our slightly modified version (with integer instead of floating-point arithmetic and inlined determinant calculation to challenge our fine-grained arithmetic decomposition optimization) is given in Lst. 2.

Listing 2: Line-Line intersection in C. The point (`INT_MAX`, `INT_MAX`) is used as a sentinel value for no intersection.

```c
typedef struct {
    int x;
    int y;
} Point;

typedef struct {
    Point s;
    Point e;
} Line;

Point LineLineIntersect(
    int sx1, int sy1, // Line 1 start
    int ex1, int ey1, // Line 1 end
    int sx2, int sy2, // Line 2 start
    int ex2, int ey2) // Line 2 end
{
    Point result;

    // Determinant(sx1, sy1, ex1, ey1);
    int detL1 = sx1 * ey1 - sy1 * ex1;
    int detL2 = sx2 * ey2 - sy2 * ex2;

    int sx1mex1 = sx1 - ex1;
    int sx2mex2 = sx2 - ex2;
    int sy1mey1 = sy1 - ey1;
    int sy2mey2 = sy2 - ey2;

    int x_nom = detL1 * sx2mex2
              - sx1mex1 * detL2;
    int y_nom = detL1 * sy2mey2
              - sy1mey1 * detL2;
    int denom = sx1mex1 * sy2mey2
              - sy1mey1 * sx2mex2;

    if (denom == 0) {
        result.x = INT_MAX;
        result.y = INT_MAX;
        return result;
    }

    result.x = x_nom / denom;
    result.y = y_nom / denom;

    return result;
}

Point mpc_main(Line INPUT_A, Line INPUT_B) {
    return LineLineIntersect(
        INPUT_A.s.x, INPUT_A.s.y,
        INPUT_A.e.x, INPUT_A.e.y,
        INPUT_B.s.x, INPUT_B.s.y,
        INPUT_B.e.x, INPUT_B.e.y);
}
```