HyCC: Compilation of Hybrid Protocols for Practical Secure Computation

Niklas Büscher
Technische Universität Darmstadt
buescher@seceng.informatik.tu-darmstadt.de

Daniel Demmler
Technische Universität Darmstadt
demmler@encrypto.cs.tu-darmstadt.de

Stefan Katzenbeisser
Technische Universität Darmstadt
skatzenbeisser@acm.org

David Kretzmer
Technische Universität Darmstadt
david.k@posteo.de

Thomas Schneider
Technische Universität Darmstadt
schneider@encrypto.cs.tu-darmstadt.de

ABSTRACT

While secure multi-party computation (MPC) is a vibrant research topic and a multitude of practical MPC applications have been presented recently, their development is still a tedious task that requires expert knowledge. Previous works have made first steps in compiling high-level descriptions from various source descriptions into MPC protocols, but only looked at a limited set of protocols.

In this work we present HyCC, a tool-chain for automated compilation of ANSI C programs into *hybrid* protocols that efficiently and securely combine multiple MPC protocols with optimizing compilation, scheduling, and partitioning. As a result, our compiled protocols are able to achieve performance numbers that are comparable to hand-built solutions. For the MiniONN neural network (Liu et al., CCS 2017), our compiler improves performance of the resulting protocol by more than a factor of 3. Thus, for the first time, highly efficient hybrid MPC becomes accessible for developers without cryptographic background.

CCS CONCEPTS

• **Security and privacy** → **Privacy-preserving protocols**; • **Software and its engineering** → *Compilers*;

KEYWORDS

secure multi-party computation; MPC; secure computation; compiler; hybrid protocols; automatization

ACM Reference Format:
Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. 2018. HyCC: Compilation of Hybrid Protocols for, Practical Secure Computation. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3243734.3243786

1 INTRODUCTION

First considered a purely theoretic result, secure multi-party computation (MPC) over arithmetic and Boolean circuits became practical in the last decade and emerged as a powerful tool to realize privacy-enhancing technologies. Recently, many new protocols with continuously improved efficiency have been proposed, e.g., Furukawa et al. [20] showed how billions of gates can be evaluated per second, and also "classic" protocols, such as Yao's garbled circuits [51] or GMW [23], have seen significant algorithmic and practical optimizations, e.g., [2, 5, 15, 25, 28, 47, 53]. It has also been shown that *hybrid* protocols, i.e., protocols that mix multiple different MPC protocols, are desirable because they outperform standalone protocols in various settings [18, 24, 32, 38, 43]. For example, for an application that consists of numerical computation and a combinatorial problem, it is beneficial to evaluate the former part with an arithmetic circuit-based MPC protocol and the latter part with a Boolean circuit-based protocol.

The continuous increase in the number of protocols and optimizations has led to a significant improvement in the performance of MPC, yet also has the drawback that MPC becomes harder and harder to access for people outside the field. Identifying a (near) optimal choice of MPC protocols for a desired application requires experience with different MPC protocols, their optimizations, their programming models, and the conversion costs to securely switch between protocols when performing a hybrid computation. Furthermore, for realizing an actual application not only expert knowledge in MPC, but also substantial knowledge in hardware design is needed to implement the application in an efficient Boolean and/or arithmetic circuit representation, which are the most common function representations in MPC. Consequently, creating efficient applications by hand is a tedious and error-prone task and therefore multiple compilers have been proposed, which share similarities with high-level synthesis from the area of hardware design.

Previous MPC compilers either only targeted a single class of protocols, e.g., Yao's garbled circuits [26, 39, 48], the GMW protocol [9, 17], or linear secret-sharing-based MPC [7], or the compilers required the developer to use specific annotations to mark which protocol is used for each statement, e.g., [18, 24]. The only other compiler that addresses the compilation of a program using two MPC protocols (Yao's gabled circuits and arithmetic sharing) is *EzPC* [14]. However, EzPC only provides semi-automation for a domain specific language (DSL), as the input code has to be manually decomposed, array accesses have to be manually resolved into multiplexer structures, and the compiled circuits are left unoptimized.

Moreover, EzPC supports only two MPC protocols, which are selected statically and independently of the execution environment, by following a strict set of rules for each expression in the program.

Compilation for hybrid MPC. In this work, we propose a novel hybrid circuit compiler, named HyCC, that is capable of compiling and optimizing applications written in standard ANSI C code into a combination of MPC protocols. In contrast to previous work, we present a fully automated approach that decomposes the source code, translates the decomposed code into Boolean and arithmetic circuits, optimizes these circuits, and finally selects suitable MPC protocols for a given deployment scenario, optimizing the selection for a given criterion, such as latency (minimal total runtime), throughput (minimal per-operation runtime), or communication.

Fig. 1 illustrates the two major components of this approach. The first component is the (one-time) compilation of the input source code into a decomposed program description in form of circuits. We refer to the different parts of a decomposed program, i.e., the compact logical building blocks a larger application consists of, as *modules*. Each module is compiled into multiple circuit representations. Our implementation compiles arithmetic circuits (A), depth-optimized circuits for GMW (B), and size-optimized circuits for Yao's protocol (Y). The second component in HyCC is the protocol selection step in which the most suitable combination of MPC protocols is selected for a decomposed program depending on the computational environment. We note that this protocol selection can be part of an MPC framework and does not necessarily need to be performed during compilation.

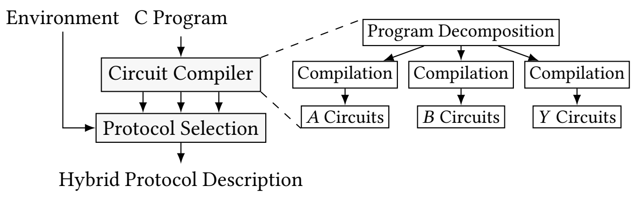


Figure 1: High-level overview of our compilation architecture. The circuit compiler decomposes an input program and compiles each part into multiple circuit representations. The protocol selection recombines the different parts.

Optimizing circuit compiler. MPC is still significantly slower and more expensive than generic plaintext computation in terms of both computation and communication. Thus, a tool-chain is required that optimizes the compilation of a program description into an efficient MPC protocol and its corresponding circuits. Even though the optimization of an input program has limits, i.e., an inefficient algorithmic representation cannot automatically be translated into a fast algorithm, a programmer expects the compiler to not only translate every statement of a high-level description of an application or algorithm for a selected target architecture, but also to optimize the given representation, e.g., by removing unnecessary computations. This is of special interest for MPC compilers, because code optimization techniques that are too expensive to be applied in traditional compilers become affordable when considering the trade-off between compile time and evaluation costs of the program on the circuit level. For example, in Yao's protocol a 32×32 bit signed integer multiplication requires the evaluation of $\approx 1{,}000$ non-linear Boolean gates (when using the best known circuit), which results in $\approx 5{,}000$ symmetric encryptions during the protocol run. Consequently, the removal of any unnecessary operation in MPC is more vital than in traditional compilation, where only a single CPU cycle is lost during program execution. We also observe that optimization techniques performed on the source code level, e.g., constant propagation, are cheaper in computational resources than minimization techniques applied on the gate level after the compilation to circuits.

These observations are reflected in our compiler architecture: Before decomposing the input source code into different parts, a rigorous static analysis is performed to realize constant propagation, detect parallelism, and determine the granularity of decomposition. The optimization then continues on the circuit level, where logic optimization techniques are gradually applied. To achieve a scalable and optimizing compilation, we guide the logic optimization efforts based on the results of static analysis of the source code. For example, loop bodies with a large number of iterations will be optimized with more effort than a piece of code that is only rarely used. Thus, in contrast to classic logic optimization or arithmetic expression rewriting, we make use of the structural information given by the programmer in the high-level code.

Summarizing the compiler's functionality, HyCC is capable of compiling optimized Boolean and arithmetic circuits suiting the requirements of most constant- and multi-round MPC protocols. Our tool-chain is highly flexible and independent of the underlying MPC protocols, as only the respective cost models for primitive operations, e.g., addition or Boolean AND, have to be adapted to reflect future protocol developments in MPC.

Protocol selection. Protocol selection is the task of mapping each part of a decomposed program to a MPC protocol representation. The circuits created by our compiler for each module and the mapping of modules into MPC protocols is sufficient to evaluate an application in a hybrid MPC framework. Optimal protocol selection is an optimization problem, where the best mapping is identified in regard to the cost model that considers the cost to evaluate each circuit in the respective MPC protocol as well as the conversion costs between the different representations. The concept of protocol selection has previously been studied independently from compilation in [32, 44]. Kerschbaum et al. [32] investigated protocol selection for a combination of Yao's garbled circuits and additive homomorphic encryption. They conjectured that the optimization problem is NP-hard and proposed two heuristic approaches. First, they presented a transformation of the combinatorial optimization problem into an integer linear programming task by linearization of the cost model. Second, they presented a greedy optimization algorithm, which is capable of optimizing larger functionalities. Pattku et al. [44] used similar heuristics to optimize the protocol selection for minimal cloud computing costs, i.e., the price to pay a cloud provider to perform a computation, as a cost model.

We follow an approach that is different in multiple aspects. First, we show that the synthesis of an efficient hybrid MPC protocol is not only a protocol selection problem, but also a scheduling problem. Second, in contrast to the work mentioned above, we make use of structural information in the source code before its translation into circuits. By grouping expressions that perform similar operations, e.g., loops, it becomes possible to perform an exhaustive search over the problem state for many practically relevant applications. Applications that cannot be optimized to the full extent with the available optimization time are approached by a combination of exhaustive search with heuristics.

Finally, by separating compilation and protocol selection, an optimized selection can be determined during protocol deployment, which allows to adapt to the given deployment scenario. For this purpose, we implement a probing technique, which evaluates the computational power and network capabilities, for precise cost estimation during protocol selection.

Our Contributions. We make the following contributions:

- We present the first complete tool-chain that automatically creates partitioned circuits and optimizes their selection for hybrid MPC protocols from standard ANSI C code, which makes hybrid MPC accessible to non-domain experts.
- We contribute techniques and heuristics for efficient decomposition of the code, scalable compilation, and protocol selection.
- We propose to separate compilation from protocol selection. This allows us to introduce a probing technique for MPC protocol implementations to optimize the protocol selection at runtime for the actual deployment scenario.
- We report speed-ups for our automatically compiled hybrid protocols of more than one order of magnitude over standalone protocol compilers, and factor three over previous handmade protocols for an exemplary machine learning application [38].

Outline. The paper is organized as follows: In §2 we discuss preliminaries of MPC and related work. Our compilation architecture is presented in §3, followed by a discussion of protocol selection and partitioning in §4. An evaluation and comparison of HyCC with related work is given in §5. We conclude in §6.

2 PRELIMINARIES AND RELATED WORK

In this section we provide a brief overview of the foundations of HyCC and related work from the field.

2.1 MPC Protocols

For simplicity reasons, in this work, we focus on MPC with two parties, also known as secure two-party computation, with security against semi-honest (passive) adversaries. Yet, we remark that the ideas and our compiler are a first starting point and can be extended to computations with more than two parties and stronger adversarial models with some engineering effort. MPC started as a field of research in the 1980s with Yao's garbled circuits protocol [51] and the protocol of Goldreich, Micali, and Wigderson (GMW) [23]. A comparison of both protocols is given in [47].

In Yao's garbled circuits [51], two parties interactively evaluate a garbled version of a Boolean circuit. One party, called garbler, creates the garbled circuit and encodes its inputs according to the garbled circuit. The other party, called evaluator, receives the garbled circuit as well as encodings of its inputs via oblivious transfer (OT) [2, 30, 42]. The evaluator then evaluates the circuit gate by gate to compute the encoding of the output, which is finally decoded. Yao's protocol has only a constant number of communication rounds and the complexity stems from the total number of AND gates in the circuit, as XOR gates can be evaluated for free [33]. Other state-of-the-art optimizations of garbled circuits that are used in today's implementations are point-and-permute [4], fixed-key AES garbling [5], and half-gates [53].

In the GMW protocol [23], XOR-secret-sharing is used to hide intermediate values. A Boolean circuit is interactively evaluated on the secret-shared data. Similar to Yao's protocol, XOR gates can be evaluated for free. AND gates require one round of communication between the parties and multiplication triples [3], that can be precomputed using OTs [2]. Thus, the complexity results from both the total number of AND gates in the circuit as well as the multiplicative depth of the circuit, i.e., the maximum number of AND gates on the critical path from any input to any output.

Arithmetic sharing works similar to the GMW protocol and uses modular addition to secret-share arithmetic values in $\mathbb{Z}_{2^\ell}$ for a bit length ℓ. Addition can be done for free, while multiplication requires one round of interaction and arithmetic multiplication triples, that can be efficiently precomputed using OTs [18, 22].

The aforementioned protocols protect the privacy of the processed data by sharing it between two parties either by using secret-sharing (GMW and arithmetic sharing) or garbling and evaluating a Boolean circuit (Yao's garbled circuits). Thus, we also refer to data processed in the respective protocols as *sharings*.

While all three sharings allow private computations, they have certain properties that make them preferable in certain scenarios. Arithmetic sharing and GMW strongly depend on low-depth circuits and a low network latency to perform well. However, they do not require symmetric cryptographic operations in the online phase, which makes them better-suited for weaker devices than Yao's protocol. Yao's protocol on the other hand is beneficial if the latency between the two parties is high, since it only requires a low constant number of communication rounds. When using arithmetic sharing, additions and multiplications are very efficient, while bit-operations require expensive bit-decomposition and are thus preferably realized with a Boolean circuit-based protocol.

Private conversion between these sharings was shown to be very efficient and can be implemented using OT [18]. However, the conversion time is non-negligible and needs to be considered when determining the overall protocol cost.

Determining an *efficient* combination of protocols for a given optimization goal is a challenging task. It depends on the use case and its complexity, the available hardware, and network connection between the parties. We automate this process and describe it in §4.

2.2 MPC Frameworks

Multiple frameworks provide practical implementations of MPC, e.g., Fairplay [39] and FastGC [28, 25] presented the first implementations of Yao's garbled circuits. Sharemind [7], SPDZ [16], and [36] are frameworks for secret sharing over arithmetic circuit-based MPC protocols. Moreover, frameworks have been proposed for different deployment scenarios, e.g., outsourcing [11, 12, 13], mobile devices [27], or to combine oblivious RAM with MPC [37].

Yet, to the best of our knowledge, only a few support hybrid MPC protocols. TASTY [24] combines Yao's garbled circuits with additively homomorphic encryption. The ABY framework [18] is a more recent framework that provides state-of-the-art implementations of Yao's garbled circuits, the GMW protocol, and arithmetic sharing for arithmetic circuits, as well as efficient conversions between these three protocols (see §2.1) in the 2-party setting. ABY^3 [40] is a novel framework for hybrid secure 3-party computation with an honest majority. The circuits generated from HyCC can directly be used by ABY and ABY^3.

2.3 Compilation for MPC

When presenting Fairplay [39], the authors realized the need for compilation for MPC, and thus not only provided a protocol implementation but also a circuit compiler for a domain specific language (DSL). Subsequently, multiple Boolean circuit compilers have been presented that improved the compilation result by using logic minimization techniques (e.g., TinyGarble [17, 48]), improved the compiler's scalability (e.g., KSS [35], Obliv-C [52], and Frigate [41]), targeted standard programming languages (e.g., CBMC-GC [26], and PCF [34]), or provided a formally verified tool-chain (e.g., CircGen [1]). Examples for arithmetic circuit compilers that target linear secret sharing-based MPC protocols are the Sharemind compiler [7] for a DSL and the PICCO compiler [54] for C.

TASTY [24] was the first hybrid protocol compiler creating circuits for its own framework (cf. §2.2), where the programmer has to manually select the respective protocol per operation. Very recently, the authors of [14] proposed a solution for hybrid compilation of MPC protocols called EzPC. However, while their main motivation is similar to ours, our results differ in several key points. In EzPC, a developer needs to invest much more work to *manually* split the input program into suitable modules and needs to *manually* resolve private array accesses into multiplexer-like structures, which hardly goes beyond what's already possible using the underlying ABY framework. Furthermore, EzPC does not apply circuit optimizations and does not consider depth-optimized Boolean circuits, as required for an efficient execution with the GMW protocol in low-latency networks.

2.4 The CBMC-GC Compiler

Our work on hybrid compilation is built on top of the compiler CBMC-GC [26], which provides a tool-chain from a (comparably) high-level language, i.e., ANSI C, to Boolean circuits. CBMC-GC has recently been extended to not only optimize circuits for size, described in [8], but also for depth [9], which jointly with its powerful symbolic execution, makes it a perfect candidate for hybrid compilation. Building on top of CBMC-GC, HyCC considers the complete input code as private functionality, i.e., public computations that are performed locally by each party are not supported. The compilation of mixed-mode languages, i.e., languages that combine public and private computation, is discussed for example in [37, 45]. We remark that common compilers like LLVM or gcc are not directly suited for MPC compilation, as they target register-based processing architectures. The creation of circuits requires a bit-precise transformation of the input source into circuits, which is closer to high-level synthesis used in hardware design.

I/O notation. In MPC the only available inputs and outputs of a program are the *input/output (I/O)* wires of the circuit. To realize the I/O mapping between C code and circuits, CBMC-GC uses a special naming convention.

For example, the source code of the millionaires' problem is given in Listing 1. The function shown is a standard C function, where only the input and output variables are specifically annotated as designated input of party A (Line 2) or party B (Line 3), or as common output for both parties (Line 4). We note that outputs can also be assigned to specific parties only. Aside from this naming convention arbitrary computations described in ANSI C are allowed, excluding floating-point operations.

```
1 void millionaires_problem() {
2   int INPUT_A_income;    // Input of Party A
3   int INPUT_B_income;    // Input of Party B
4   int OUTPUT_result = 0; // Output to both Parties
5
6   if (INPUT_A_income > INPUT_B_income) {
7     OUTPUT_result = 1;
8   }
9 }
```

Listing 1: CBMC-GC [26] code example for Yao's Millionaires' problem.

3 THE HYCC MPC COMPILER

Here we describe our hybrid compiler[1]. After introducing the challenges, we provide details on every step of the compilation chain.

3.1 Hybrid Compilation and its Challenges

We begin with a description of a straight-forward (unoptimized) approach to compile hybrid MPC protocols from standard source code in order to illustrate the challenges of achieving efficient hybrid compilation. We will then refine this approach throughout this section and describe a more advanced compilation approach.

An exemplary illustration of the necessary steps for a straight-forward compilation is given in Fig. 2. First, the input source code is decomposed into multiple parts, henceforth referred to as *modules*. Modules are the finest level of granularity used in the later protocol selection. Thus, all code within a module is guaranteed to be evaluated with the same MPC protocol. We remark that during protocol evaluation this level of granularity is only forming a lower bound. In principle, a program can also be evaluated with only a single MPC protocol. The decomposition can be made directly on the source code level or on an intermediate representation of the code, e.g., Single Static Assignment (SSA) form. Given a decomposed application description, each module is compiled into the circuit representations for the different MPC protocols forming the hybrid protocol and then optimized. In this work, we consider size-optimized Boolean circuits required for Yao sharing (Y), depth-optimized Boolean circuits required for GMW style protocols (B), and arithmetic circuits (A). Finally, the hybrid protocol is synthesized during protocol selection and scheduling (cf. §4).

Multiple challenges (besides the complexity of compiling efficient Boolean or arithmetic circuits itself) arise when following this

[1]An open source implementation will be made available at `https://gitlab.com/securityengineering/HyCC`.

```
 1 long pow(unsigned b, unsigned exp) {
 2     /* Computationally expensive code */
 3 }
 4
 5 void main(){ /* Some code */
 6     t1 = pow(x, y);
 7     t2 = pow(2, y);
 8     /* Some code */
 9     unsigned c = 1;
10     if (condition)
11         c += 1;
12     res = pow(x, c);
13     /* Some code */
14 }
```

Listing 2: Example source code to illustrate the conflict between local and inter-procedural optimization.

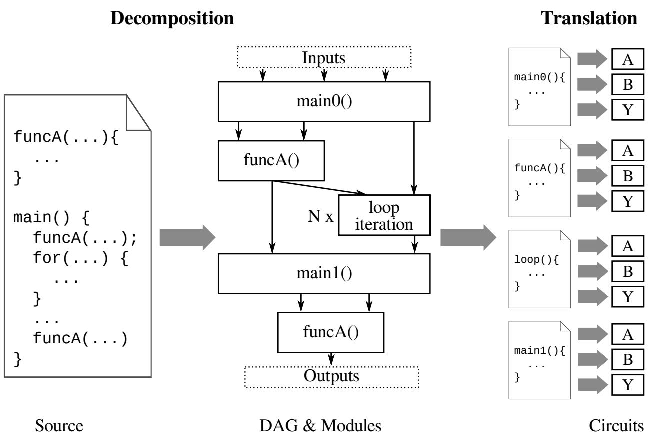


Figure 2: Naïve compilation of hybrid protocols from input source code to a decomposed circuit description. First, the code is decomposed into multiple modules. Then, each module is translated into three different circuit formats.

straight-forward approach. All challenges relate to a trade-off between compilation resources, i.e., time and storage, and compilation result, i.e., circuit size and depth. We describe identified challenges and propose solutions, which motivate our actual compilation architecture:

- *Granularity of decomposition.* Automatically decomposing input code into multiple modules is a non-trivial task, as a fine-grained decomposition limits the possibility of circuit level optimizations and increases the complexity of the computationally expensive protocol selection problem, whereas a coarse-grained decomposition risks to miss the most efficient selection. We tackle this challenge by the use of heuristics based on static analysis of the source code.
- *Local versus inter-procedural optimization.* Optimizing an application as a whole or optimizing its modules independently can lead to circuits of different sizes. The former allows more optimizations, whereas the latter is typically more efficient w.r.t. compilation because each module will only be compiled and optimized once.
  We illustrate this conflict with the example in Listing 2. This example consists of a function `main()` that performs multiple calls to a function `pow()`, which computes the power of two integers. A function-wise decomposition approach would separate the two functions to compile them independently. However, a careful study of the source code reveals that the `pow()` function is called with a constant argument in Line 7, and with the second argument being either one or two in Line 12, which simplifies the computation of the exponentiation function on the circuit level significantly. An optimizer with an inter-procedural (context-sensitive) or holistic view could detect this fact and optimize the created circuit accordingly. To find a trade-off between modular and holistic optimization, i.e., compile time and circuit size, we rely on static analysis and source code optimization techniques in our compilation framework.
- *Loop handling.* Loops are an essential part of many programs. To create circuits with low complexity, it is best to first unroll (inline) all loop iterations, before translating them into a circuit, as this allows to apply optimizations, such as constant propagation, over all iterations. However, for compilation efficiency, for the exploitation of parallelism, and for a more compact circuit representation, it can be useful to avoid loop unrolling. Therefore, instead of choosing either technique we propose an adaptive approach that distinguishes different loop types and then decides for or against loop unrolling.
- *Efficient logic minimization.* Even though we consider compilation to be a one-time task, which in theory allows to use arbitrary resources, in practice compilation efficiency is of relevance. Optimizing circuits on the gate-level is a resource-consuming task that can become practically infeasible when considering circuits with Billions of gates. Therefore, we adapt and improve a technique referred to as *Source-guided optimization* [10] to optimize circuits under configurable time constraints by distributing an optimization budget in a controlled manner.

The sketched solutions can be realized using static source code analysis techniques only. This is sufficient because MPC applications have to be bound (finite and deterministic runtime), as they are evaluated independently of the program's input to avoid any form of information leakage. Using the side-channel free circuit computation model, all possible program paths are visited during protocol runtime and thus can already be studied at compile time.

3.2 Architecture

We describe our compilation architecture for a resource-constrained environment that expects a source code with a pointer to an entry function f as input, and a compilation and optimization time limit T. The compiler outputs a program description consisting of multiple modules, compiled to different circuit representations, and a direct acyclic dependency graph that describes the dependencies between the different modules. The combination of dependency graph and modules can be used to evaluate the program in a hybrid MPC framework.

The compilation architecture consists of multiple compilation *phases* shown in the next paragraph, which themselves can consist of multiple compilation *passes*. The phases are:

(1) *Automated Parallelization (§3.2.1):* Automated identification of code blocks that can be evaluated in parallel using external tools.
(2) *Preprocessing, Lexing and Parsing (§3.2.2):* Construction of an Abstract Syntax Tree (AST) from the input code.
(3) *Source Code Optimization and Loop Unrolling (§3.2.3):* Source-to-source compilation using static analysis.
(4) *Code Decomposition (§3.2.4):* Decomposition of the input program into multiple modules.
(5) *Circuit Compilation (§3.2.5):* Compilation of each module in the different circuit representations.
(6) *Inter-Procedural Circuit Optimization (§3.2.6):* Optimization of Boolean and arithmetic circuits across multiple modules.
(7) *Circuit Export (§3.2.7):* Writing the decomposed circuit to a file, ready for reconstruction in protocol selection.

Note that steps 2, 3, and 5 are also part of CBMC-GC's original tool-chain [26], whereas the others have been added for the compilation of hybrid protocols. We describe the steps in detail in the following subsections.

3.2.1 *Automated Parallelization.*

Parallel code segments allow efficient compilation and protocol selection. Moreover, most MPC protocols profit from parallelized functionalities. Therefore, their detection is of relevance in compilation for hybrid MPC. Due to the availability and maturity of automated parallelization tools, e.g., [29, 50], we rely on these for the detection of *parallel loops*, i.e., loops that have independent loop iterations. These tools are able to detect parallelism and to annotate parallelism using source-to-source compilation techniques, independent of the HyCC compilation chain. For annotations, HyCC relies on the OpenMP notation, which is the de-facto application programming interface for shared memory multiprocessing programming in C and supported by most parallelization tools. Specific preprocessing notations, e.g., `#omp parallel for`, are added in the code line before each parallel loop. The annotations are parsed in the next compilation phase.

3.2.2 *Preprocessing, Lexing and Parsing.*

The preprocessing, lexing, and parsing of source code is realized as in CBMC-GC [26]. We remark that, as in CBMC-GC and typical for MPC, the given program has to be bound to avoid leaking information through the program runtime. Furthermore, global variables are not supported, which, however, is an implementation limitation and not a limitation of our approach. The only difference between HyCC and CBMC-GC is that the annotated parallelism is parsed.

3.2.3 *Source Code Optimization and Loop Unrolling.*

In this compilation step the intermediate code is analyzed and optimized using static analysis. The results are subsequently used as a preparation step for the later code decomposition and parallelization. In detail, to overcome the optimization limits of (context-insensitive) modular compilation, rigorous source code optimization in form of a partial evaluation is performed. Thus, all variables known to be *constant* are propagated, such that every remaining expression (indirectly) depends on at least one input variable (*dynamic variable*).

To achieve an efficient compilation result, partial evaluation requires a symbolic execution of the complete source code, which limits compilation scalability. A faster compile time can be achieved, under a (often significant) circuit-size trade-off, when not optimizing across function or loop boundaries. For example, the circuit compiler Frigate [41] follows this approach. To achieve the best of both worlds, we propose a time-constrained multi-pass optimization routine, which can be interrupted at any point in time. Given sufficient compile time, the iterative approach converges to the same result as a holistic optimization.

In the first pass, partial evaluation is only performed with a local scope, yet not across function or loop boundaries. In the second pass, constants are propagated within every function body and between multiple functions (*inter-procedural constant propagation*), yet not between multiple loop iterations or in recursive function calls to avoid loop unrolling. This form of program specialization can lead to an increase in the code size (function cloning), as the same function may now appear multiple times with different signatures. For example, in Listing 2, we observe that the `pow()` function is called with none, either of the two, and both arguments being constant. Hence, in this example, two, namely one with the first argument and one with the second argument being constant, additional copies of the function will be introduced, partially evaluated, and compiled individually.

In the third optimization pass, all (possibly nested) loops are visited. We distinguish three types of loops: Parallel, simple, and complex loops. *Parallel* loops have already been identified in the first compilation phase. We refer to a `for` loop as *simple* if the loop guard is constant and the iterator variable is incremented (or decremented) in a constant interval and not written inside the loop body. Furthermore, simple loops cannot have `return` or `break` statements. Hence, the loop range of simple loops can be derived without a complete symbolic execution of the loop itself. *Complex* loops are all remaining loops, which require a complete unrolling of all iterations using symbolic execution to determine their termination.

Simple and parallel loops do not need to be unrolled during compilation, as it is sufficient to compile a single circuit for all iterations that are instantiated multiple times within an MPC protocol with the loop iterator variable as input. Nevertheless, similar to function specialization, loop specialization is desirable for an efficient compilation result. Therefore, in HyCC, loops are optimized in an iterative approach. First, all constants that are independent of the loop iterator variable are propagated in the loop body. This allows an effective optimization of multiple loop iterations at the same time. Afterwards, the first iteration of every loop is partially evaluated. In contrast to the previous symbolic execution, the loop iterator variable is now initialized with a constant and can lead to further program specialization. If symbolic execution of the first iteration leads to improvements, i.e., an expression can be evaluated or removed, then the loop becomes a candidate for unrolling. By unrolling the first loop iteration, an estimate on the computational resources required to unroll all iterations can be made. Given sufficient remaining compile-time (and memory), the loop will be unrolled and optimized.

Function and loop specialization may reveal constants relevant for other code parts. Therefore, given sufficient remaining compile-time, a further round of partial evaluation is initiated until no further improvements are observed. Finally, a call-graph is exported for usage in the following decomposition. Statements within loops that have been unrolled are enriched with information about their

```c
unsigned scalar = x1 * y1 + x2 * y2;
if(scalar > min) {
  count = count + 1;
}
```

Listing 3: Code excerpt to illustrate code decomposition. The scalar product of two two-dimensional vectors is computed and compared to a reference value.

original position in the loop, to re-identify loops and their iterations during decomposition.

3.2.4 Code Decomposition. Identifying a suitable decomposition is the major challenge for efficient protocol partitioning. The task of automated decomposition is to identify which parts of a code should jointly be compiled as one module, which forms the finest level of granularity of protocol selection. Each module has an input and an output interface, where a module can receive input from one or more modules and provide output to one or more modules. We refer to the separation points between two modules as interface. Hence, a decomposed code forms a directed acyclic graph (DAG) consisting of modules with interfaces in-between (similar to a call-graph or dependency graph). The first input and last output interface of the graph are the program input and output variables, respectively.

The overall goal of a good decomposition heuristic is to identify modules of a program that can be evaluated efficiently in a specific circuit representation. A first example of such a heuristic are expressions consisting only of arithmetic statements. Naturally, these should profit from processing in MPC protocols based on arithmetic circuits. In contrast, control flow operations or comparisons are evaluated more efficiently with Boolean circuit-based protocols. Consequently, arithmetic and combinatorial statements should be in different modules. We follow a multi-pass decomposition approach that starts with the complete source code as a module that is split into more fine-granular modules in every pass.

Function decomposition. Functions already give programs a form of modularization and hence they can be used as natural boundaries for decomposition. Therefore, in the first compilation pass, each function (considering the function specialization described in §3.2.3) becomes a module. The input interface to a function module consists of the arguments that are read in the function body and assigned to other variables. The output interface are all pointers and variables passed by reference that are written to in the function body, as well as the `return` statement. This form of recursive decomposition leads to three modules per (possibly nested) function call, one module for the callee itself, one for the code before and one after the function call.

Technically, this decomposition becomes challenging when pointers or references are passed to a function. Using the results of the previous (exhaustive) symbolic execution, input and output variables can be differentiated, and array sizes can be determined during compile time. We note that dynamic memory management, i.e., memory that is allocated based on (private) input variables, is impossible to be realized in the circuit computation model and is thus outside the scope of circuit compilers.

Loop decomposition. Loops also give code a structure and are therefore a good heuristic for decomposition. Consequently, in the second compilation pass, every module is further decomposed according to its loops, such that every loop iteration becomes its own module, where all variables that are read from an outer scope and the iterator variable form the input interface and all variables that are written to, but defined in an outer scope, form the output interface.

Loops might have been unrolled during code optimization (cf. §3.2.3). For their re-identification during decomposition, loop iterations are marked as such during loop unrolling. Loops that have not been unrolled during code optimization require a dedicated handling of array accesses before decomposition, i.e., a pointer analysis. Otherwise, array accesses that depend on the iterator variable, which is an input variable after decomposition, would compile into private array accesses that are of significant circuit size [26]. For better efficiency, in HyCC these array accesses are extracted from the loop iteration and placed in the module that encapsulates the iteration. Consequently, these array accesses are evaluated as accesses with publicly known index, and as such without any gates.

Decomposition by loops is especially beneficial for parallel loops, as it allows to derive the placement costs of MPC protocols during protocol selection from the analysis of only one loop iteration.

Arithmetic decomposition. In the last decomposition pass, connected arithmetic expressions are extracted, as they are candidates for arithmetic circuits. Therefore, all expressions in each module are visited to extract expressions that purely consists of arithmetic operations (supported by the used MPC protocol). This decomposition is realized as follows: For each module, a data flow dependency graph is constructed from the output to the input interface. Each node in the dependency graph is an elementary expression and an edge represents the data that is computed on. By iterating over all nodes, two sets of sub-graphs are formed. The first contains sub-graphs consisting of connected arithmetic expressions, whereas the second contains sub-graphs consisting of connected remaining expressions. Each sub-graph forms its own module, where edges between the sub-graphs define the respective I/O interfaces. This form of decomposition is illustrated in Fig. 3 for the code excerpt given in Listing 3 that computes a scalar multiplication.

We remark that during protocol selection, multiple (or even all) modules can be merged to larger modules, that are jointly evaluated with the same MPC protocol. Finally, the created DAG that represents the modules and their I/O dependencies is exported for the next compilation steps.

3.2.5 Circuit Compilation. The different modules identified in the previous step are compiled separately into two or three circuit representations. Namely, every module is compiled into size-optimized Boolean circuits using the circuit compiler of CBMC-GC and into depth-optimized Boolean circuits using its ShallowCC extension [9], which uses depth-optimized building blocks (cf. §2.4). Moreover, every module that can be represented with the arithmetic operations supported by ABY (cf. §2.2) is also compiled into an arithmetic circuit using a straight-forward mapping of arithmetic expressions to arithmetic gates. Note that modules representing functions or loops that have not been unrolled are only compiled once.

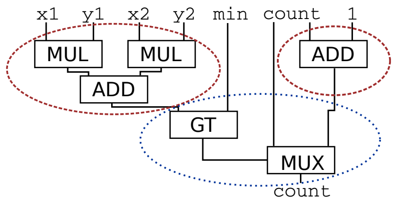


Figure 3: Code dependency graph and decomposition for the code excerpt in Listing 3. Connected statements that can efficiently be expressed as arithmetic circuits, i.e., addition (ADD) and multiplication (MUL), marked with a red dashed circle, form sub-graphs. Statements that profit from a Boolean representation, marked with a blue dotted circle, i.e., greater-than (GT) and multiplexer (MUX), are grouped.

3.2.6 Inter-Procedural Circuit Optimization. So far, the compiled circuits have only be optimized on the source code level, cf. §3.2.3. After their creation, all Boolean circuits are optimized on the gate-level, e.g., by removing unused bits (gates). Constants are propagated between different modules and circuit types, including the arithmetic circuits when applicable.

The scalability of logic minimization techniques for Boolean circuits is limited, because these techniques are applied in a gate-by-gate manner and some techniques involve computationally expensive operations, such as SAT sweeping to identify unnecessary gates [8]. Thus, to distribute the available computational resources onto all modules efficiently, we adapt a technique called source-code guided optimization [10]. Here, over multiple time-constrained optimization passes, the available computing time is distributed using the structural information present on the source-code level and the information about previous optimization passes. Namely, in each pass, the available budget, which is a fraction of the total upper compile time specified by the user, is distributed according to the unrolled circuit size of each module. Thus, modules originating from loops or function bodies are optimized with an effort that is proportional to the number of their iterations or calls. Moreover, outputs of modules identified as constant are propagated to subsequent modules and optimization preference is given to modules with many constant input gates.

In contrast to [10], which did not target hybrid compilation, we optimize all types of circuits independently with a shared optimization budget. Thus, the different optimizations are performed separately. Yet, the information about identified constant output variables or module outputs is propagated between optimization routines, and will also be used to improve the arithmetic circuits, if all bits of an output variable are identified as constant. We remark that this form of cross propagation maintains functional correctness because all circuits for one module are logically equivalent.

3.2.7 Circuit Export. Once a user-defined compile time has been reached, the optimization routine is stopped and the DAG, consisting of modules with optimized circuit representations and I/O interfaces as well as information about identified parallel loops, is exported. Given the circuits, an MPC framework can choose a protocol selection to perform the computation as described next.

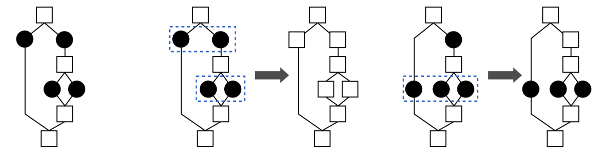
4 PROTOCOL SELECTION AND SCHEDULING

In this section, we describe how to determine an optimized scheduling and mapping of the modules that were created during compilation to MPC protocols.

4.1 Problem Definition

We optimize evaluation costs of a hybrid MPC application by choosing an efficient protocol representation and evaluation order of all modules for a given program description. For this, we present heuristics considering a user-specified cost model. A very interesting use case is the optimization of the protocol's online runtime, yet, various other cost models are also of interest. For example optimizing the cloud computing costs, i.e., the price to pay for computational power and traffic to perform a computation using cloud service providers, has been discussed in [44]. Further examples are the total protocol runtime including or excluding the time spent on preprocessing depending on the use case of the application, the pure communication costs when considering a constrained network connection, or the power consumption when considering mobile devices. All these minimization problems can also be formulated as constrained problems, e.g., minimizing the communication costs while keeping the protocol runtime below a user-defined threshold.

The computation and communication costs of a hybrid MPC protocol depend on the combined costs to evaluate each module in the selected protocol plus the time to convert between modules, when evaluating them with different protocols. However, in contrast to previous works, i.e., [32] and [44], we observe that the optimization problem, i.e., achieving minimal costs for a given decomposition, is not only a protocol selection problem but also a scheduling problem. Namely, the evaluation order of parallel modules, i.e., modules without sequential dependencies, can significantly influence the effectiveness of protocol selection, and thus the overall protocol runtime. This is because of the non-linearity of computation and communication costs of MPC protocols (e.g., parallel computations in the program can be performed in the same communication round or packed in the same cryptographic operation), as well as the trade-off that has to be taken into account when converting between different MPC protocols. Fig. 4 illustrates this scheduling problem for an example program description and naïve cost model. Namely, Fig. 4a shows an exemplary program DAG resembling a computation from inputs (top) to outputs (bottom) with different modules (nodes) in between. For simplicity, we assume that modules illustrated as squares profit from an evaluation in a sharing type A (e.g., arithmetic), whereas modules represented by circles profit from a different sharing type B (e.g., Boolean). Furthermore, for illustration purposes, we assume that a conversion between two different sharings is reducing the total evaluation costs if at least three modules are evaluated in the same sharing. The result of an exemplary as-soon-as-possible (ASAP) scheduling followed by a protocol selection is shown in Fig. 4b. Two groups of nodes (marked with dashed lines) become a candidate for being evaluated in sharing type B. However, when considering the assumption above, an optimal protocol selection algorithm will propose to evaluate all modules with type A, as the conversion is too expensive for only two modules. An optimal scheduling is shown in Fig. 4c. In this case, three modules that can jointly be evaluated in sharing type B can be identified during protocol selection and are consequently evaluated in sharing type B.



(a) Example DAG with protocol preferences **(b) ASAP scheduling and subsequent protocol selection** **(c) Optimal scheduling and subsequent protocol selection**

Figure 4: Exemplary DAG with different evaluation schedules and protocol selections described §4.1.

Thus, we remark that optimal runtime can only be achieved when optimizing both protocol selection and scheduling of modules. Next, we present a formalization of the optimization problem, before presenting optimization routines in the following subsections.

Formalization. We formalize the cost model and optimization problem as follows. Given is a program description in the form of a DAG G from inputs $i \in In$ to outputs $o \in Out$ with modules $m \in M$ in between. Cost minimization for hybrid MPC consists of two interleaving tasks, namely protocol selection and scheduling. Protocol selection is an assignment that maps every module to an MPC protocol, also referred to as sharing type $t : M \to \{A, B, Y\}$. We denote the set of protocols representing each modules in the selected sharing with Π^t_M. Moreover, we denote the set of conversion protocols required to convert between adjacent modules evaluated with different MPC protocols with Π^t_C.

Scheduling is the task of assigning an evaluation order to all modules for a given protocol selection. Modules and their conversions form the set of elementary protocols $\Pi^t = \Pi^t_M \cup \Pi^t_C$ that are the atomic units of scheduling. As it is common in scheduling, we use the notion of instructions I, which is the set of protocols that are performed in parallel in hybrid MPC. Furthermore, note that most modules and their conversions have data dependencies to other modules, i.e., module m_2 is dependent on m_1, if the result of m_1 is needed to compute m_2. Therefore, scheduling is the task of creating a sequence of k instructions $(I_1, I_2, \ldots, I_k)$ and assigning protocols to instructions $s : \Pi^t \to I_1, \ldots, I_k$. This assignment must guarantee that every protocol only appears in one instruction, protocols in each instruction are pairwise mutually independent, and the order of protocols induced through the order of instructions confirms to the dependencies between modules and conversions.

Given a schedule, i.e., an ordered list of instructions IL, the total evaluation cost is the sum of the evaluation costs of all protocols representing a module $\pi^t_m \in \Pi^t_M$ and their respective conversions $\pi^t_c \in \Pi^t_C$ according to IL plus the cost to input, i.e., share, values into the protocol, plus the costs to reveal all outputs. In §4.3, we illustrate the computation of the evaluation cost in more detail by performing a runtime prediction for a given evaluation schedule.

In summary, the goal of optimized protocol selection and scheduling is to minimize the total evaluation cost by choosing a schedule s and protocol selection t. Next, we present approaches to achieve efficient protocol selection and scheduling.

4.2 Protocol Selection in HyCC

Scheduling and protocol selection are tightly coupled problems, where the latter alone is conjectured to be NP-hard [32, 44]. Therefore, in HyCC we first select an evaluation schedule using a heuristic for a given program decomposition. The schedule is then used in a second step to optimally solve the protocol selection problem.

Scheduling. In HyCC, protocol scheduling is performed with respect to the parallelism present on the source code level. Consequently, the identified parallelism, which has been annotated in the program's DAG during compilation, is used to schedule modules in parallel. This explicit scheduling of parallel code structures is necessary, as the straight forward application of an ASAP or other scheduling algorithm cannot guarantee that parallel code statements will be evaluated in parallel, as shown in Fig. 4. Moreover, this approach is beneficial for hybrid MPC, as the MPC protocols, conversion protocols, and their implementations benefit from parallel execution. For example, n sequentially scheduled multiplications in an arithmetic circuit require n communication rounds, whereas a parallel alignment allows to perform all multiplications in a single communication round, which leads to very different runtimes in an high-latency network. Furthermore, parallelization is beneficial for the later protocol selections, as multiple modules can be grouped together and thus, optimized more efficiently.

Besides parallelization, modules are scheduled in an ASAP manner. To combine both strategies in a single algorithm, parallel modules are merged in a single module when creating an ASAP schedule. Afterwards, the merged modules are restored and placed in the same instruction of the evaluation schedule. We leave more advanced scheduling algorithms for future work.

Protocol selection. Even though in the general case protocol selection is conjectured to be NP-hard, given a coarse-grained decomposition, such as the one created by HyCC, an optimal protocol selection can be computed under reasonable computational effort for many practical applications, as we show in §5.1. This is because the complexity of the protocol selection routine is dominated by the width of the program's DAG G rather than its size. Consequently, all applications that only moderately divert in their data and control flow are candidate problems for optimal protocol selection.

To identify the optimal protocol selection for a given DAG G, we apply a straight-forward combinatorial optimization approach by enumerating all possible protocol combinations using dynamic programming. The core concept of the optimization routine is to iteratively optimize the selection of protocols up to a certain module, following the order of modules generated by the instruction list IL. In every step, one module is added and modules that do not have any open outputs, i.e., outputs that are required for subsequent modules, are removed. We refer to the set of modules with open outputs as the working set WS. For every WS, the best protocol selection for every possible sharing combination is computed and stored. When going from WS to the next WS', the best protocol selection to represent the new WS' in every sharing combination is computed by identifying the least cost to compute WS' from any configuration of WS. Thus, the complexity of this optimization approach for a given DAG G with n modules, a maximum width of w, and s different sharing types is in $O(ns^w)$, and thus exponential in the size of the largest working set, i.e., the width of G. Consequently, for a small number of sharing types and for DAGs with moderate width, the protocol selection problem can be solved optimally in seconds, as evaluated in §5.1.

```
eval_costs($WS, WS', c', cost\_table$)
1 :  $min \leftarrow \infty$
2 :  foreach $c \in$ share_combinations($WS$)
3 :    $cost \leftarrow$ cost_step($WS$, $WS'$, $c$, $c'$)
4 :    if $cost + cost\_table[c] < min$ then
5 :      $min \leftarrow cost + cost\_table[c]$
6 :    endif
7 :  return $min$
8 :  endfor
```

Algorithm 2: Algorithm to compute the cheapest evaluation cost to compute the next WS' in a specific sharing configuration. The algorithm takes as input the two working sets WS, WS', the designated sharing configuration for WS' denoted with c', as well as a table with the cheapest cost to compute all possible sharing configurations c of WS.

```
protocol_selection(DAG $G$, instruction list $IL$)
1 :  $WS \leftarrow G.inputs$
2 :  foreach $c$ in share_combinations($WS$) do
3 :    $cost\_table[c] \leftarrow$ cost_input_sharing($c$)
4 :  endfor
5 :  foreach $m$ in $G.modules$ ordered by $IL$ do
6 :    $WS' \leftarrow$ remove_completed($WS \cup m$)
7 :    foreach $c' \in$ share_combination($WS'$) do
8 :      $cost\_table'[c'] \leftarrow$ eval_costs($WS$, $WS'$, $c'$, $cost\_table$)
9 :    endfor
10 :    $WS \leftarrow WS'$
11 :    $cost\_table \leftarrow cost\_table'$
12 :  endfor
13 :  return min($cost\_table$)
```

Algorithm 1: Optimal protocol selection algorithm. The algorithm takes as input the DAG of the program with the circuit descriptions of all modules. It returns the protocol cost for the optimal protocol selection.

Algorithmic implementation. The protocol selection algorithm is given in Algorithm 1 and described next. The initial WS consists of all inputs of the DAG G. Consequently, the cost to represent a WS in a specific sharing combination is the cost to share each input with the specified sharing (Line 2). Next, the iterative optimization routine is initiated. A module from the ordered G is added to the WS and completed modules are removed to create the next working set WS' (Line 6). Then, all possible sharing combinations of the next WS' are enumerated. For each of these combinations, the best protocol selection based on all sharing configurations of the previous WS is computed (Line 8). This task is realized in function eval_costs(), outlined in Algorithm 2, which takes as input the two working sets, as well as the desired sharing configuration c' of WS' and a cost table that stores the costs to compute all possible configurations of WS. The costs to evaluate the newly added module, reflecting the sharings in WS and WS' is computed in function cost_step(), which models the evaluation costs of MPC protocols. A discussion on cost models is given in §4.3. Once all possible sharings of WS' are computed, WS is replaced by WS' to add a further module. The algorithm ends, once all modules have been visited, and thus an optimal output sharing has been identified.

Scalable protocol selection. In cases where the DAG exceeds the computationally manageable width, the optimization algorithm can compute the optimal protocol selection for all sub-graphs, which have a width that is solvable. For the remaining sub-graphs, or the combination of multiple sub-graphs, heuristics, such as the hill-climbing heuristic proposed in [32] could be used to search for an optimized selection in the combination of different optimally solved sub-graphs.

4.3 Cost Model and Probing

The most relevant cost factors of MPC are the total protocol runtime, the bandwidth requirement and the number of communication rounds between the MPC parties. An accurate cost model is required for an optimized protocol selection. The total communication complexity can precisely be predicted by summing the communication costs of all individual building blocks of a hybrid protocol, whereas the runtime prediction is more complex. For large circuits (i.e., several million non-linear gates) a simple approach can give a rough estimate, where the circuit depth d is multiplied by the communication latency T_{lat} and added with the number of non-linear gates G_{nl} divided by the maximum throughput of non-linear gates per second TP_{nl} to get a runtime estimate $T_{\text{estim}} = d \cdot T_{lat} + G_{nl}/TP_{nl}$. However, this does not work for smaller circuits, as these do not fully saturate the network connection. We follow a more complex approach, where the input of the runtime prediction is the computation and communication costs of the individual protocol building blocks, i.e., input and output sharing, AND and XOR gates, arithmetic addition and multiplication gates, share conversions, as well as the available computation and communication resources. We automatically measure runtime, required communication and circuit depth, i.e., round complexity, of each individual building block for different input sizes and all available sharing types. Moreover, we evaluate them with different degrees of parallelism, to consider the efficiency gain of parallelization and also to determine the limits of the available resources. To optimize for the best possible performance, this probing takes place on the systems where the final hybrid protocol will be deployed. By doing this, we can provide an estimate for the runtime and bandwidth requirement of the compiled hybrid MPC protocols without actually running them by linear inter- and extrapolation of the previously measured smaller building blocks.

Fig. 5 shows a comparison of empirically measured runtimes (solid lines) and estimated runtimes (dashed lines) for three use cases: Minimum-Euclidean-Distance (described in §5.2.1) and AES evaluated with Yao's garbled circuits and the GMW protocol, respectively. We benchmark the building blocks for different input sizes, which are evaluated in parallel. Extrapolating from runtime that was measured on small building blocks to a full-sized circuit and the influence of the network connection between the MPC parties leads to imprecision in the runtime prediction. In our measurements we found that the prediction was always within −50% and +50% of the actual achieved runtime. For better runtime prediction a larger

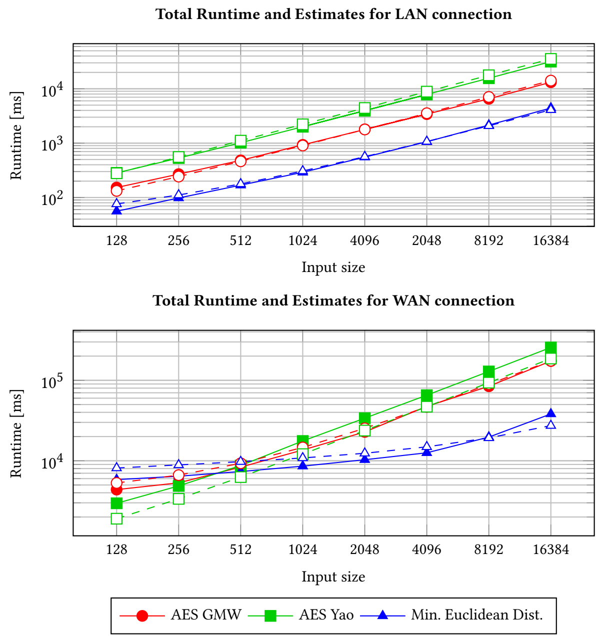


Figure 5: Comparison of measured runtimes (solid lines) and corresponding estimates (dashed lines) using a log-log plot.

number of measurements and more data points of the underlying building blocks are required to limit the influence of noise on a busy network or on shared hardware. As our results show, the runtime estimate that is interpolated from measuring the underlying building blocks captures the relative runtime between the protocols well and allows for identifying the most efficient sharing in the protocol selection step for a given deployment scenario.

5 BENCHMARKS

In this section, we present an experimental evaluation of HyCC. We study the efficiency of protocol selection, the circuits created by HyCC, and their performance in hybrid MPC protocols for various use cases in two different deployment scenarios. The goal of this evaluation is to illustrate that the circuits that were automatically created by HyCC from ANSI C code are comparable to hand-crafted hybrid circuits and significantly more efficient than previous single-protocol compilers. As such, we are able to show that HyCC is simplifying the ease-of-use of hybrid MPC, and is thus a powerful tool to prototype a solution for a privacy problem, which allows to identify whether generic MPC protocols achieve sufficient efficiency or whether dedicated protocols need to be developed. We remark that the goal of this work is not to outperform dedicated secure computation protocols, which are optimized to achieve maximum efficiency for a specific use case. We begin with an evaluation of the runtime of the protocol selection algorithm presented in §4.2.

5.1 Protocol Selection

To illustrate that an exhaustive search is a sufficient solution for the protocol selection problem in most practical cases, we measure the runtime of the protocol selection algorithm in Fig. 6. Shown are the runtimes averaged over $k = 10$ executions of a straight forward (unoptimized) implementation running on a commodity laptop for randomly generated graphs with $n = 2 \cdot w$ modules and increasing graph width w. We observe the expected exponential growth in runtime when increasing w. Albeit being a limiting factor of our approach, to the best of our knowledge all applications in privacy research studied so far have a very small branching factor in their functionalities, which leads to very small width w. For example, all use cases in this work have a width of at most $w = 3$, which is solved in less than 0.01 seconds and we remark that even larger graphs with a width of $w = 10$ are solved in seconds.

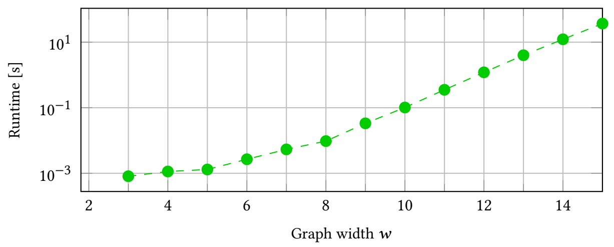


Figure 6: Runtime of the protocol selection algorithm for different graph widths w.

5.2 Use Cases

Next, we evaluate the generated circuits and protocol selections made by HyCC for different use cases in the ABY framework [18]. The ABY framework provides state-of-the-art implementations for Yao's protocol, GMW, OT-based additive secret sharing protocols, and the corresponding conversion protocols, which makes it an ideal backend to evaluate the created circuits. For the evaluation, we use applications that illustrate the versatility of HyCC or that have previously been used to benchmark MPC protocols and compilers.

Experimental setup. All applications are implemented based on textbook algorithms and compiled with HyCC using a total optimization time of $T = 10$ minutes. The generated circuits are evaluated on two identical machines with an Intel Core i7-4790 CPU and 32 GiB RAM, connected via a 1 Gbps local network, denoted as LAN. To simulate an Internet connection between the MPC parties, denoted as WAN, we use the Linux tool `tc` to set a latency of 50 ms (100 ms RTT) and limit the throughput to 100 Mbps. We set the symmetric security parameter to 128 bit. Running times are median numbers from 10 measurements. "—" denotes that no values were given or benchmarked.

For all applications the number of non-linear (multiplicative) gates, communication rounds, transferred bytes, and the protocol runtime of the setup phase and of the online phase are measured. For comparison purposes we provide these numbers not only for the best protocol selection but also for different instantiations of the same functionality, e.g., all modules evaluated in a Boolean circuit-based protocol, or a hybrid of a Boolean circuit and arithmetic sharing. As before, we use A for arithmetic sharing, B for Boolean sharing using the GMW protocol, and Y for Yao's garbled circuits. We omitted A-only measurements for use cases that include bit-operations (e.g., minimum, comparison), since these are extremely costly in A sharing and therefore not implemented in ABY [18].

```c
#define N 128
#define D 2

#include <inttypes.h>
typedef int32_t DT;

DT match(DT db1, DT db2, DT s1, DT s2) {
  DT dist1 = db1 - s1;
  DT dist2 = db2 - s2;
  return dist1 * dist1 + dist2 * dist2;
}

void mpc_main() {
  DT INPUT_A_db[N][D];
  DT INPUT_B_sample[D];
  DT matches[N];

  DT min = match(INPUT_A_db[0][0], \
     INPUT_A_db[0][1], INPUT_B_sample[0], \
     INPUT_B_sample[1]);

  for(int i = 1; i < N; i++) {
    DT dist = match(INPUT_A_db[i][0], \
     INPUT_A_db[i][1], INPUT_B_sample[0], \
     INPUT_B_sample[1]);

    if(dist < min) {
      min = dist;
    }
  }
  DT OUTPUT_res = min;
}
```

Listing 4: Biometric matching code example.

Table 1: Modules and their circuit sizes when compiling the biometric matching example with HyCC.

Module	non-linear gates			non-lin.	# bits	
	A	B	Y	depth	Inputs	Outputs
mpc_main	0	0	0	0	8,256	32
match (×128)	2	1,785	1,536	20	128	32
loop1 (×127)	—	120	64	10	64	32
total	128	$2 \cdot 10^5$	$2 \cdot 10^5$	909	8,256	32

5.2.1 *Biometric Matching (Minimum Euclidean Distance).* The minimum Euclidean distance is the minimum of the distances from a single coordinate to a list of coordinates. It is used in biometric matching between a sample and a database, and is a well-known benchmark for MPC, e.g., [9, 18, 24]. For illustration purposes a code example for the biometric matching functionality is shown in Listing 4 for a database of size $n = 128$ and dimension $d = 2$. The identified modules and their circuit sizes when compiling this code with HyCC are given in Tab. 1.

For the experimental evaluation we use databases consisting of $n \in \{1{,}000; 4{,}096; 16{,}384\}$ samples with dimension $d = 4$, where each coordinate has bit length $b = 32$ bits. The performance results are given in Tab. 2. We compare a hand-built hybrid ABY circuit [18] with a circuit that is compiled with HyCC. The results show that the circuits that we automatically compiled from a standard ANSI C description achieve the same complexity as the circuits that were hand-built and manually optimized in ABY. Here, a combination of arithmetic sharing and Yao's protocol (A+Y) achieves the best runtime in all settings. The runtimes in both implementations show a slight variation that is due to variance of the network connection. We remark that the setup phase of the ABY circuit is more efficient, because ABY allows single-instruction-multiple-data (SIMD) preprocessing, which is currently not implemented in HyCC.

To show the efficiency gain of hybrid protocols over standalone protocols, we give experiments using B or Y sharing only. These protocols are significantly less efficient and for larger input sizes even exceed the memory resources of our benchmark hardware.

5.2.2 *Machine Learning.* Machine learning (ML) has many applications and is a very active field of research. Protecting the privacy of training data or ML inputs is also an active research area.

Supervised machine learning – Neural networks. Deep (Convolutional) Neural Networks (CNNs) are one of the most powerful ML techniques. Therefore many dedicated protocols for private data classification using CNNs have been proposed recently [21, 38, 46]. We implemented CryptoNets [21] and the very recent MiniONN CNN [38], which both have been proposed to detect characters from the MNIST handwriting data set. Previously these use cases needed to be carefully built by hand, while we achieve even better performance when conveniently compiling easily understandable C source code to a hybrid MPC protocol.

Tab. 3 shows machine learning performance results. For Cryptonets, HyCC automatically determined A as the best sharing in the LAN setting. When changing the activation function (from the square function to $f(x) = max(0, x)$, known as RELU function), or when changing the number representation (fixed-point instead of integer), a hybrid A+Y protocol becomes the fastest option.

For the MiniONN CNN, HyCC proposes to use A+Y, where Y is mainly used to compute the RELU activation function, which results in a hybrid protocol that requires only a third of the online runtime, total runtime, and total communication compared to the original MiniONN protocol [38]. When expressing the entire MiniONN functionality solely as a Boolean circuit, more than 250 million non-linear gates are used. Using Yao's protocol in the LAN setting, sending the corresponding garbled circuit would take more than one minute, assuming perfect bandwidth utilization. Thus, in comparison to all existing Boolean circuit compilers for MPC, i.e., single protocol compilers, HyCC achieves a runtime that is more than one order of magnitude faster.

Unsupervised machine learning – k-means. Clustering is another data mining task, frequently used to identify centroids in unstructured data. One of the most well known clustering algorithms is k-means, and multiple works proposed dedicated privacy-preserving k-means protocols, e.g., [31, 49]. We evaluate a textbook algorithm that detects $c = 4$ clusters in 2-dimensional data sets of size $n = 500$ using $i = 8$ iterations and show our results in Tab. 3. Also in this use case, a hybrid A+Y protocol achieves the best runtime.

5.2.3 *Gaussian elimination.* Solving linear equations is required in many applications with Gaussian elimination being the most well known solving algorithm. We implement a textbook Gauss solver with partial pivoting for $n \in \{10, 16\}$ equations using a fixed-point number representation and present results in Tab. 4. Fixed-point numbers can be implemented in software, and thus also in HyCC, with only a few lines of code, which is illustrated in Appendix A. In all scenarios, HyCC identifies A+Y as the most efficient protocol, where Y is mainly used to compute the row permutations and divisions. Note that due to the significant circuit depth, we did not

Table 2: Minimum Euclidean distance benchmarks comparing a hand-built circuit (ABY [18]) with a compilation from HyCC (best values marked in bold).

				Setup Phase			Online Phase		
Circuit	Sharing	non-linear Gates	Interaction Rounds	LAN [ms]	WAN [ms]	Comm. [MiB]	LAN [ms]	WAN [ms]	Comm. [KiB]
min. Euclid ABY [18] (n = 1,000)	A+Y	98,936	6	167	2,878	8	**55**	**557**	**1,567**
min. Euclid HyCC (n = 1,000)	A+Y	98,936	10	175	1,920	8	70	584	1,582
min. Euclid ABY [18] (n = 1,000)	A+B	155,879	78	151	2,206	9	73	3,971	1,620
min. Euclid HyCC (n = 1,000)	A+B	155,879	80	190	3,622	10	131	4,249	1,643
min. Euclid HyCC (n = 1,000)	Y	3,166,936	3	1,498	10,239	99	1,177	1,789	4,016
min. Euclid HyCC (n = 1,000)	B	3,497,879	93	550	8,228	107	2,932	7,974	1,725
min. Euclid ABY [18] (n = 4,096)	A+Y	405,440	6	420	7,336	34	**211**	**1,234**	**6,416**
min. Euclid HyCC (n = 4,096)	A+Y	405,440	10	536	5,162	34	330	1,406	6,480
min. Euclid ABY [18] (n = 4,096)	A+B	638,855	92	417	8,016	37	303	5,606	6,629
min. Euclid HyCC (n = 4,096)	A+B	635,020	94	555	4,337	41	689	5,802	6,722
min. Euclid HyCC (n = 16,384)	A+Y	1,621,952	10	2,239	13,522	112	1,419	4,041	25,920
min. Euclid HyCC (n = 16,384)	A+B	2,540,935	108	2,286	15,179	164	3,155	11,024	26,883

Table 3: Machine learning nenchmarks comparing with MiniONN [38] and CryptoNets [21] (best values marked in bold).

				Setup Phase			Online Phase		
Circuit	Sharing	non-linear Gates	Interaction Rounds	LAN [ms]	WAN [ms]	Comm. [MiB]	LAN [ms]	WAN [ms]	Comm. [KiB]
MiniONN MNIST [38]	—	—	—	3,580	—	21	5,740	—	651,877
MiniONN MNIST HyCC	A+B	2,275,880	90	1,750	14,469	165	2,689	9,443	35,864
MiniONN MNIST HyCC	A+Y	1,838,120	34	1,825	14,041	150	**1,621**	**5,882**	**35,094**
CryptoNets Square [21]	—	—	—	0	—	0	297,500	—	381,133
CryptoNets Square HyCC	A	107,570	7	683	10,348	131	**134**	1,359	**2,018**
CryptoNets RELU HyCC	A+Y	195,455	19	784	11,238	134	163	1,297	3,330
CryptoNets RELU HyCC	A+B	195,455	33	735	11,298	134	187	1,917	3,360
CryptoNets Fix-Point HyCC	A+B	195,455	33	765	11,416	134	187	1,910	3,694
CryptoNets Fix-Point HyCC	A+Y	195,455	19	780	11,264	134	162	**1,296**	3,330
k-means HyCC (n = 500)	A+B	7,894,592	6,578	3,453	21,887	293	5,917	337,083	**30,473**
k-means HyCC (n = 500)	A+Y	4,991,816	125	4,414	21,007	206	**3,748**	**10,503**	38,915

Table 4: Gaussian elimination benchmarks (best values marked in bold).

				Setup Phase			Online Phase		
Circuit	Sharing	non-linear Gates	Interaction Rounds	LAN [ms]	WAN [ms]	Comm. [MiB]	LAN [ms]	WAN [ms]	Comm. [KiB]
Gauss 10 × 10 HyCC	A+B	555,611	41,305	340	—	29	5,843	—	2,989
Gauss 10 × 10 HyCC	B	1,158,995	41,829	268	—	23	6,020	—	1,412
Gauss 10 × 10 HyCC	A+Y	494,215	147	348	2,849	17	**256**	4,235	1,997
Gauss 10 × 10 HyCC	Y	1,030,225	3	561	3,850	31	429	**631**	**101**
Gauss 16 × 16 HyCC	A+B	2,516,310	67,920	1,245	—	57	11,182	—	10,031
Gauss 16 × 16 HyCC	A+Y	2,294,615	243	1,515	8,842	79	**1,258**	8,126	7,740
Gauss 16 × 16 HyCC	Y	4,393,173	3	2,445	13,749	134	1,957	**2,190**	**257**

measure the runtime for Boolean circuits evaluated with the GMW protocol in the WAN setting.

5.2.4 *Database analytics.* Performing data analytics on sensitive data has numerous applications and therefore many privacy-preserving protocols and use cases have been studied, e.g., [6, 19]. Using generic MPC techniques is of interest for database analytics, as it allows to perform arbitrary analytics, e.g., hypothesis testing, or allows to add data perturbation techniques, e.g., differential privacy, before releasing the result with minimal effort. We study exemplary use cases, where each party provides a database (array) of size n_A and n_B that has two columns each, which are concatenated (merged), leading to a database of size $n = n_A + n_B$, or joined (inner join on one attribute) yielding a database of maximum size $n = n_A \cdot n_B$, and then the mean and variance of one column of the combined database are computed. The performance evaluation is shown in Tab. 5. We observe that in both use cases, a combination of A+Y achieves minimal runtime in the LAN setting, with the division (and join) being performed in Y. In the WAN setting, Y achieves optimal runtime and minimal online communication.

5.2.5 *Summary of Experiments.* Summarizing the results obtained in all use cases, we observe that hybrid protocols consisting of A+Y, achieve very efficient runtime in the LAN deployment, whereas Y is often the fastest protocol in the WAN deployment. We observe that the GMW protocol (B) has barely been identified to achieve optimal

Table 5: Database operation benchmarks (best values marked in bold).

				Setup Phase			Online Phase		
Circuit	Sharing	non-linear Gates	Interaction Rounds	LAN [ms]	WAN [ms]	Comm. [MiB]	LAN [ms]	WAN [ms]	Comm. [KiB]
DB Merge 500 + 500 HyCC	B	1,441,732	1,237	593	3,776	44	1,310	63,430	733
DB Merge 500 + 500 HyCC	A+B	5,395	1,187	29	927	2	144	59,319	56
DB Merge 500 + 500 HyCC	Y	849,711	3	858	3,619	26	679	886	752
DB Merge 500 + 500 HyCC	A+Y	4,990	17	22	815	1	**4**	**606**	**30**
DB Join 50 × 50 HyCC	B	4,429,046	765	1,645	13,312	135	4,219	43,090	2,179
DB Join 50 × 50 HyCC	A+B	529,526	708	451	5,201	26	564	36,652	6,827
DB Join 50 × 50 HyCC	Y	2,550,076	3	1,725	8,317	78	1,272	**1,451**	**100**
DB Join 50 × 50 HyCC	A+Y	443,900	32	472	3,433	23	**435**	2,395	6,705
DB Join 25 × 200 HyCC	B	8,981,870	767	3,521	26,766	274	9,846	48,937	4,403
DB Join 25 × 200 HyCC	A+B	1,163,575	708	832	7,085	54	1,202	38,155	13,295
DB Join 25 × 200 HyCC	Y	5,158,825	3	3,212	15,960	158	2,660	**2,861**	**250**
DB Join 25 × 200 HyCC	A+Y	937,049	32	927	5,837	47	**942**	3,603	12,861

runtime for any of the benchmark applications. This is because we performed all benchmarks in the function dependent preprocessing model, which is the default setting in ABY, and which allows to garble the circuit in the setup phase. When using a function independent cost model for preprocessing, HyCC identifies *A+B* as the fastest protocol combination in the LAN setting for many applications.

6 CONCLUSIONS AND FUTURE WORK

In our evaluation we observed that hybrid protocols can significantly outperform standalone protocols. HyCC is capable of automatically synthesizing the required hybrid protocols from a high-level description and selecting them for a given deployment scenario. As such, HyCC is even capable of outperforming certain hand-optimized protocols. Moreover, as the manual creation of circuits and their selection are tedious and error-prone tasks, we conclude that HyCC makes hybrid MPC more practical and also accessible to developers without expert-knowledge in MPC.

In future work, we will extend HyCC with floating point operations and integrate more MPC protocols with different cost models. A natural candidate for extension is homomorphic encryption, similar to TASTY [24]. Another possibility would be integrating trusted hardware environments such as Intel's SGX.

ACKNOWLEDGMENTS

We thank all anonymous reviewers for their helpful and constructive comments. This work has been co-funded by the German Federal Ministry of Education and Research (BMBF) and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP, by the DFG as part of projects E4 and S5 within the CRC 1119 CROSSING, and by the DFG as part of project A.1 within the RTG 2050 "Privacy and Trust for Mobile User".

REFERENCES

[1] J. B. Almeida et al. Jasmin: high-assurance and high-speed cryptography. In *ACM CCS'17*, pp. 1807–1823. ACM Press, 2017.
[2] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *ACM CCS'13*, pp. 535–548. ACM Press, 2013.
[3] D. Beaver. Correlated pseudorandomness and the complexity of private computations. In *ACM STOC'96*, pp. 479–488. ACM Press, 1996.
[4] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols (extended abstract). In *ACM STOC'90*, pp. 503–513. ACM Press, 1990.
[5] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE S&P'13*, pp. 478–492. IEEE Computer Society Press, 2013.
[6] D. Bogdanov, M. Jõemets, S. Siim, and M. Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *FC'15*, pp. 227–234. Springer, 2015.
[7] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: a framework for fast privacy-preserving computations. In *ESORICS'08*, pp. 192–206. Springer, 2008.
[8] N. Büscher, M. Franz, A. Holzer, H. Veith, and S. Katzenbeisser. On compiling boolean circuits optimized for secure multi-party computation. *Formal Methods in System Design*, 51(2):308–331, 2017.
[9] N. Büscher, A. Holzer, A. Weber, and S. Katzenbeisser. Compiling low depth circuits for practical secure computation. In *ESORICS'16*, pp. 80–98. Springer, 2016.
[10] N. Büscher, D. Kretzmer, A. Jindal, and S. Katzenbeisser. Scalable secure computation from ANSI-C. In *IEEE International Workshop on Information Forensics and Security (WIFS'16)*, pp. 1–6. IEEE, 2016.
[11] H. Carter, C. Lever, and P. Traynor. Whitewash: outsourcing garbled circuit generation for mobile devices. In *ACM ACSAC'14*, pp. 266–275. ACM Press, 2014.
[12] H. Carter, B. Mood, P. Traynor, and K. Butler. Outsourcing secure two-party computation as a black box. In *Security and Communication Networks (SCN'16)*, pp. 2261–2275. Wiley Online Library, 2016.
[13] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure outsourced garbled circuit evaluation for mobile devices. In *USENIX Security'13*, pp. 289–304. USENIX Association, 2013.
[14] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi. EzPC: programmable, efficient, and scalable secure two-party computation. Cryptology ePrint Archive, Report 2017/1109, 2017. `http://eprint.iacr.org/2017/1109`.
[15] S. G. Choi, K.-W. Hwang, J. Katz, T. Malkin, and D. Rubenstein. Secure multi-party computation of Boolean circuits with applications to privacy in on-line marketplaces. In *CT-RSA'12*, pp. 416–432. Springer, 2012.
[16] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO'12*, pp. 643–662. Springer, 2012.
[17] D. Demmler, G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, and S. Zeitouni. Automated synthesis of optimized circuits for secure computation. In *ACM CCS'15*, pp. 1504–1517. ACM Press, 2015.
[18] D. Demmler, T. Schneider, and M. Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, 2015.
[19] W. Du, Y. S. Han, and S. Chen. Privacy-preserving multivariate statistical analysis: linear regression and classification. In *SIAM International Conference on Data Mining'04*, pp. 222–233, 2004.
[20] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *EUROCRYPT'17*, pp. 225–255. Springer, 2017.
[21] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning, ICML'16*, pp. 201–210, 2016.
[22] N. Gilboa. Two party RSA key generation. In *CRYPTO'99*, pp. 116–129. Springer, 1999.

[23] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC'87*, pp. 218–229. ACM Press, 1987.

[24] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In *ACM CCS'10*, pp. 451–462. ACM Press, 2010.

[25] W. Henecka and T. Schneider. Faster secure two-party computation with less memory. In *ASIACCS'13*, pp. 437–446. ACM Press, 2013.

[26] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ANSI C. In *ACM CCS'12*, pp. 772–783. ACM Press, 2012.

[27] Y. Huang, P. Chapman, and D. Evans. Privacy-preserving applications on smartphones. In *USENIX Conference on Hot Topics in Security (HotSec'13)*, pp. 4–4. USENIX Association, 2011.

[28] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security'11*, pp. 331–335. USENIX Association, 2011.

[29] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *International Conference on Supercomputing (ICS'91)*, pp. 244–251. ACM Press, 1991.

[30] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO 2003*, pp. 145–161. Springer, 2003.

[31] G. Jagannathan and R. N. Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining'05*, pp. 593–599, 2005.

[32] F. Kerschbaum, T. Schneider, and A. Schröpfer. Automatic protocol selection in secure two-party computations. In *ACNS 14*, pp. 566–584. Springer, 2014.

[33] V. Kolesnikov and T. Schneider. Improved garbled circuit: free XOR gates and applications. In *ICALP'08*, pp. 486–498. Springer, 2008.

[34] B. Kreuter, A. Shelat, B. Mood, and K. R. B. Butler. PCF: A portable circuit format for scalable two-party secure computation. In *USENIX Security'13*, pp. 321–336. USENIX Association, 2013.

[35] B. Kreuter, A. Shelat, and C. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security'12*, pp. 285–300. USENIX Association, 2012.

[36] Y. Lindell and A. Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *ACM CCS'17*, pp. 259–276. ACM Press, 2017.

[37] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *IEEE S&P'15*, pp. 359–376. IEEE Computer Society Press, 2015.

[38] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In *ACM CCS'17*, pp. 619–631. ACM Press, 2017.

[39] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay – a secure two-party computation system. In *USENIX Security'04*, pp. 287–302. USENIX Association, 2004.

[40] P. Mohassel and P. Rindal. ABY^3: a mixed protocol framework for machine learning. Cryptology ePrint Archive, Report 2018/403, 2018. http://eprint.iacr.org/2018/403.

[41] B. Mood, D. Gupta, H. Carter, K. R. B. Butler, and P. Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *IEEE EuroS&P'16*, pp. 112–127. IEEE, 2016.

[42] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *ACM Conference on Electronic Commerce'99*, pp. 129–139. ACM, 1999.

[43] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *IEEE S&P'13*, pp. 334–348. IEEE Computer Society Press, 2013.

[44] E. Pattuk, M. Kantarcioglu, H. Ulusoy, and B. Malin. CheapSMC: A framework to minimize secure multiparty computation cost in the cloud. In *DBSec'16*, pp. 285–294. Springer, 2016.

[45] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *IEEE S&P'14*, pp. 655–670. IEEE Computer Society Press, 2014.

[46] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: a hybrid secure computation framework for machine learning applications. In *ACM ASIACCS'18*. ACM Press, 2018.

[47] T. Schneider and M. Zohner. GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In *FC'13*, pp. 275–292. Springer, 2013.

[48] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar. TinyGarble: highly compressed and scalable sequential garbled circuits. In *IEEE S&P'15*, pp. 411–428. IEEE Computer Society Press, 2015.

[49] J. Vaidya and C. Clifton. Privacy-preserving *k*-means clustering over vertically partitioned data. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining'13*, pp. 206–215. ACM Press, 2003.

[50] R. P. Wilson et al. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, 1994.

[51] A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *IEEE FOCS'86*, pp. 162–167. IEEE Computer Society Press, 1986.

[52] S. Zahur and D. Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. http://eprint.iacr.org/2015/1153.

[53] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *EUROCRYPT'15*, pp. 220–250. Springer, 2015.

[54] Y. Zhang, A. Steele, and M. Blanton. PICCO: a general-purpose compiler for private distributed computation. In *ACM CCS'13*, pp. 813–826. ACM Press, 2013.

A FIXED-POINT COMPUTATIONS IN HYCC

In Listing 5 a code example is shown that implements 32 bit fixed-point numbers for ANSI C that can be used in any application compiled with HyCC.

```
#include <inttypes.h>

#define FP_BITS 32
#define FP_INTEGER_BITS 24
#define FP_FRACTION_BITS (FP_BITS - FP_INTEGER_BITS)

typedef int32_t fixedpt;
typedef int64_t fixedptd;

fixedpt fixedpt_mul(fixedpt a, fixedpt b)
{
  return  ((fixedptd)a * (fixedptd)b)  >>  \
             (fixedptd)FP_FRACTION_BITS);
}

fixedpt fixedpt_div(fixedpt a, fixedpt b)
{
  return ((fixedptd)a<<(fixedptd)FP_FRACTION_BITS)/b;
}
```

Listing 5: Code to add fixed-point support in ANSI C and thus, also applications compiled with HyCC.