Automated Synthesis of Optimized Circuits for Secure Computation

Daniel Demmler
TU Darmstadt, Germany
daniel.demmler@ec-spride.de

Ghada Dessouky
TU Darmstadt, Germany
ghada.dessouky@cased.de

Farinaz Koushanfar
Rice University, USA
farinaz@rice.edu

Ahmad-Reza Sadeghi
TU Darmstadt, Germany
ahmad.sadeghi@cased.de

Thomas Schneider
TU Darmstadt, Germany
thomas.schneider@ec-spride.de

Shaza Zeitouni
TU Darmstadt, Germany
shaza.zeitouni@cased.de

ABSTRACT

In the recent years, secure computation has been the subject of intensive research, emerging from theory to practice. In order to make secure computation usable by non-experts, Fairplay (USENIX Security 2004) initiated a line of research in compilers that allow to automatically generate circuits from high-level descriptions of the functionality that is to be computed securely. Most recently, TinyGarble (IEEE S&P 2015) demonstrated that it is natural to use existing hardware synthesis tools for this task.

In this work, we present how to use industrial-grade hardware synthesis tools to generate circuits that are not only optimized for size, but also for depth. These are required for secure computation protocols with non-constant round complexity. We compare a large variety of circuits generated by our toolchain with hand-optimized circuits and show reduction of depth by up to 14%.

The main advantages of our approach are developing customized libraries of depth-optimized circuit constructions which we map to high-level functions and operators, and using existing libraries available in the industrial-grade logic synthesis tools which are heavily tested. In particular, we show how to easily obtain circuits for IEEE 754 compliant floating-point operations. We extend the open-source ABY framework (NDSS 2015) to securely evaluate circuits generated with our toolchain and show between 0.5 to 21.4 times faster floating-point operations than previous protocols of Aliasgari et al. (NDSS 2013), even though our protocols work for two parties instead of three or more. As application we consider privacy-preserving proximity testing on Earth.

Keywords

secure computation; automation; optimization, logic design; hardware description

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
CCS'15, October 12–16, 2015, Denver, CO, USA.
© 2015 ACM. ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813678.

1. INTRODUCTION

Secure computation allows multiple parties to evaluate a function on their private inputs without revealing any information except for the result of the computation. The first protocols given were Yao's garbled circuits protocol [Yao86] and the protocol of Goldreich-Micali-Wigderson (GMW) [GMW87]. Both protocols securely evaluate a Boolean circuit that represents the desired functionality. Since then, a large body of literature has been investigating the design and implementation of practical circuit-based secure computation in different adversarial settings. While designing efficient and correct circuits for smaller building blocks for simple applications can be performed manually by experts, this task becomes highly complex and time consuming for large applications such as floating-point arithmetic and signal processing, and is thus error-prone. Faulty circuits could potentially break the security of the underlying applications, e.g., by leaking additional information about a party's private inputs. Hence, an *automated* way of generating *correct* large-scale circuits which can be used by regular developers is highly desirable.

A large number of compilers for secure computation such as [MNPS04, BNP08, HKS$^+$10, HEKM11, Mal11, MLB12, KSS12, HFKV12, SZ13, KSMB13, ZSB13] implemented circuit building blocks manually. Although tested to some extent, showing the correctness of these compilers and their generated circuits is still an open problem.

Recently, TinyGarble [SHS$^+$15] took a completely different approach by using already established powerful hardware logic synthesis tools and customizing them to be adapted to automatically generate Boolean circuits for functions to be evaluated by Yao's garbled circuits protocol. The advantage of this approach lies in the fact that these tools are being used by industry for designing digital circuits, and hence are tested thoroughly, which is justified by the high production costs of Application-Specific Integrated Circuits (ASICs). However, these tools are designed primarily to synthesize circuits on hardware target platforms such as ASICs or configurable platforms such as Field Programmable Gate Arrays (FPGAs) or Programmable Array Logic (PAL). Using hardware logic synthesis tools for special purposes such as generating circuits for secure computation, requires customizations and workarounds. Exploiting these tools promises accelerated and automated circuit generation, significant speedup, and ease in designing and generating circuits for much more complicated functions, while also maintaining the size (and depth) efficiency of hand-optimized smaller circuit building blocks. In particular, TinyGarble exploited the sequential logic to synthesize highly compact circuits. However, TinyGarble considered only few functionalities: addition, Hamming weight, comparison, multiplication, matrix multiplication, AES, SHA-3, and a MIPS CPU.

In this work we continue along the lines of using logic synthesis tools for secure computation and automatically synthesize an extensive set of basic and complex operations, including IEEE 754 compliant floating-point arithmetic. In contrast to TinyGarble, which generated size-optimized circuits for Yao's garbled circuits protocol, we focus on synthesizing depth-optimized circuits for the GMW protocol [GMW87]. Although the round complexity of the GMW protocol depends on the circuit depth, it has some advantages compared with Yao's constant-round protocol: 1) it allows to precompute *all* symmetric cryptographic operations in a setup phase and thus offers a very efficient online phase, 2) its setup phase is independent of the function being computed, 3) it balances the workload equally between all parties, 4) GMW allows for better parallel evaluation of the same circuit (SIMD operations) [SZ13, DSZ15], 5) it can be extended to multiple parties, and 6) the TinyOT protocol [NNOB12] which provides security against stronger active adversaries, has an online phase which is very similar to that of GMW, and its round complexity also depends on the circuit depth.

We combine industrial-grade logic synthesis tools with the recent open-source ABY framework [DSZ15] which implements state-of-the-art optimizations of the two-party protocols by GMW and Yao. On the one hand, our approach allows to use existing and tested libraries for complex functions such as IEEE 754 compliant floating-point operations that are already available in these tools without the need to re-implement them manually. On the other hand, this allows to use high-level input languages such as Verilog where we map high-level operations to our optimized implementations of basic functions.

1.1 Outline and Our Contributions

After summarizing related work in §1.2 and preliminaries in §2, we present our following contributions:

Architecture and Logic Synthesis (§3). We provide a fully-automated end-to-end toolchain allowing the developer to describe the function to be computed securely in a high-level Hardware Description Language (HDL), such as Verilog, followed by the generation of the required customized circuit and its secure evaluation using either GMW [GMW87] or Yao's protocol [Yao86]. Our toolchain uses hardware synthesis tools, both open-source and commercial, to generate depth- and size-optimized circuits customized for both protocols respectively. For this, we manipulate and engineer state-of-the-art hardware synthesis tools with synthesis constraints and customized libraries to generate circuits optimized for either protocol according to the developer's choice.

Optimized Circuit Building Blocks (§4). We develop a library of depth-optimized and size-minimized circuits, including arithmetic operations (e.g., addition, subtraction, multiplication, division), comparison, counter, and multiplexer, which can be used to construct more complex functionalities such as various distances, e.g., Manhattan, Euclidean, or Hamming distance. Some of the implemented building blocks show improvements in depth compared with hand-optimized circuits of [SZ13] by up to 14%, while others show at least equivalent results. Assembling sub-blocks from our customized library can be used to construct more complicated functionalities, which would otherwise be impossible to build and optimize by hand. We exploit the capabilities of our synthesis tools to bind high-level operators (e.g., the `'+'` operator) and functions to optimized circuits in our library to allow the developer to describe circuits in Verilog using high-level operators. We also utilize built-in Intellectual Property (IP) libraries in commercial hardware synthesis tools to generate Boolean circuits for more complex functionalities such as floating-point arithmetic which have been verified and tested extensively.

Benchmarks and Evaluation (§5). We use the ABY framework [DSZ15] to securely evaluate the Boolean circuits generated by our hardware synthesis toolchain. Moreover, we extend the list of available operations in ABY by multiple floating-point operations. In contrast to previous works that built dedicated and complex protocols for secure floating-point operations, we use highly tested industrial-grade floating point libraries. We compare the performance of our constructions with related work. For floating-point operations we achieve between 0.5 to 21.4 times faster runtime than [ABZS13] and 0.1 to 3 267 times faster runtime than [KW14]. We emphasize that we achieve these improvements even in a stronger setting, where all but one party can be corrupted and hence our protocols also work in a two-party setting, whereas the protocols of [ABZS13, KW14] require a majority of the participants to be honest and hence need $n \geq 3$ parties. We also present timings for integer division that outperform related work of [ABZS13] (3-party) by a factor of 0.6 to 3.7 and related work of [KSS13] (2-party) by a factor of 32.4 to 274. Additionally, we present benchmarks for matrix multiplication, but here we are slower than previous approaches of [BNTW12, ZSB13, DSZ15].

Application: Private Proximity Testing (§6). A real world application of floating-point calculations on private inputs is privacy-preserving proximity testing on Earth [ŠG14]. We implement the formulas described in [ŠG14] with our floating-point building blocks and achieve faster runtime as well as higher precision compared to their protocols. This demonstrates that our automatically generated building blocks can outperform hand-built solutions.

1.2 Related Work

We classify related work into different categories next.

TinyGarble. Most related to our work is the recently proposed TinyGarble framework [SHS$^+$15] which was the first work to consider using hardware-synthesis tools to automatically generate circuits for secure computation. The authors used sequential circuits that allow to describe a circuit as a loop over a smaller sub-circuit (e.g., an ℓ-bit ripple-carry adder can be represented as iterating ℓ times over a single bit adder). Thereby, they are capable of generating highly compact circuit descriptions. Although this approach allows to represent the circuits in a highly memory-efficient way, the total number of gates that are evaluated securely and hence the communication and total number of crypto operations remains unchanged. As the main goal of TinyGarble was to assess the memory efficiency, the paper gives benchmarks only for evaluating a single circuit, the ripple-carry adder, with Yao's garbled circuits protocol.

As described before in §1, the GMW protocol has several advantages over Yao's garbled circuits protocol (precomputation, load balancing, multiple parties, etc.), but requires circuits with low depth. Unfortunately, sequential circuits cannot directly be applied to the GMW protocol, since the sequential circuit structure can significantly increase the depth of the circuit and thus the communication rounds required by GMW. Our work is the first to consider automated hardware synthesis of low-depth combinational circuits optimized for use in the GMW protocol, as well as size-optimized circuits for Yao's protocol. Our work also allows developers to write high-level Verilog code which can be automatically mapped to our optimized circuits by binding our circuit descriptions to arithmetic operators.

Secure Computation Compilers from Domain Specific Languages. Fairplay [MNPS04, BNP08] and the compatible PAL compiler [MLB12] compile a functionality in a domain specific input language, called Secure Function Definition Language (SFDL), into a Boolean circuit described in the Secure Hardware Definition Language (SHDL) which is evaluated with Yao's garbled circuits protocol. Our intermediate circuit description is very similar to Fairplay's SHDL; in fact we could easily process SHDL input. Similarly, TASTY [HKS+10] proposed a domain specific input language called TASTYL that allows to combine protocols that mix Yao's garbled circuits with additively homomorphic encryption. The compiler presented in [KSS12] also provides a domain specific input language and showed scalability to circuits consisting of billions of gates that were evaluated with a variant of Yao's protocol with security against malicious adversaries. Recently, ObliVM [LWN+15] introduced a domain specific language that is compiled into a Yao-based secure computation protocol with support for Oblivious RAM (ORAM).

Instead of using a domain specific input language, we use existing Hardware Description Languages (HDLs) such as Verilog or VHDL that are already known by many developers. Thereby, we can use existing code and allow a large community of developers to specify functionalities without the necessity of learning a new language.

Secure Computation Compilers from ANSI C. The following secure computation tools use a subset of the ANSI C programming language as input. CBMC-GC [HFKV12] initiated this line of development and used a SAT solver to generate size-optimized Boolean circuits from a subset of ANSI C. PCF [KSMB13] compiles into a compact intermediate representation that also supports loops, similar to the sequential circuits of TinyGarble described above. Both CBMC-GC and PCF target Yao's garbled circuits protocol and hence only optimize for size. PICCO [ZSB13] is a source-to-source compiler that allows parallel evaluation and uses secure computation protocols based on linear secret sharing with at least three parties.

Although ANSI C is widely known as well, it has the drawback that some operations are either not supported (e.g., pointer arithmetic) or incur significant costs when compiled into a circuit (e.g., array access depending on private values). Thereby, existing C code sometimes needs to be rewritten or results in inefficient protocols. Although we do not eliminate these restrictions in our work, these issues do not occur when taking existing functionalities described in HDLs that do not support pointers and often avoid accesses to arrays with private indices, as these result in costly multiplexers.

Secure Computation Libraries. In this class of tools, the developer composes the circuits to be evaluated securely from circuit libraries that are instantiated at runtime. This approach has been proposed in FastGC [HEKM11] and VMCrypt [Mal11] both of which are based on Yao's garbled circuits. In fact, all implementations of the GMW protocol [CHK+12, SZ13, DSZ15] are secure computation libraries.

In our work we extend the ABY framework [DSZ15] to process pre-compiled sub-circuits that can then be composed dynamically at runtime.

2. PRELIMINARIES

In this section we provide preliminaries and background related to the GMW protocol (§2.1), hardware synthesis (§2.2), and the IEEE 754 floating-point standard (§2.3).

2.1 The GMW protocol

In the GMW protocol [GMW87], two or more parties compute a function that is encoded as Boolean circuit. The parties' private inputs and all intermediate gate values are perfectly hidden by an XOR-based secret sharing scheme. GMW allows to evaluate XOR gates locally, without interaction, using only one-time pad operations and thus essentially for free. AND gates, however, require interaction in the form of Oblivious Transfers (OTs) or Beaver's multiplication triples [Bea91] that can be pre-computed in a setup phase, which is independent from the parties' private inputs and the function being computed. This pre-computation can be achieved efficiently by using OT extension [IKNP03, ALSZ13] as shown in [CHK+12, SZ13]. After evaluating all circuit gates in the online phase, the output can be reconstructed by computing the XOR of the resulting output shares.

In order to achieve high performance, the total number of AND gates in the circuit (the circuit size $\mathbf{S}$) and the number of AND gates from any input to any output wire (the circuit depth $\mathbf{D}$) should be low. In this work we use the variant of the GMW protocol with two parties and security against passive/semi-honest adversaries.

2.2 Hardware Synthesis

Hand-optimizing Boolean circuits for secure computation is a tedious, error-prone and time-consuming task. Using hardware synthesis tools for synthesizing and optimizing these circuits, and even more complex circuits that cannot be easily hand-optimized, seems to be a promising and natural approach. As shown in TinyGarble [SHS+15], using hardware synthesis tools allows to reduce the time and effort invested by further automating the process of generating optimized Boolean netlists in terms of circuit size and/or depth.

Overview. Hardware or logic synthesis is the process of translating an abstract form of circuit description into its functionally equivalent gate-level logic implementation using a suite of different optimizations and mapping algorithms that have been a theme of research over years. A logic synthesis tool is a software which takes as input a function description (functional, behavioral or structural description, state machine, or truth table) and transforms and maps this description into an output suitable for the target hardware platform and manufacturing technology.

Tools. Common target hardware platforms for synthesized logic include Field Programmable Gate Arrays (FPGAs), Programmable Array Logics (PALs), and Application Specific Integrated Circuits (ASICs). ASIC synthesis tools, as opposed to FPGA synthesis tools, are used in this work due to the increased flexibility and options allowed in their synthesis tools, and because FPGA synthesis tools map circuits into Look-up Tables (LUTs) and flip-flop (FF) gates in accordance with FPGA architectures, and not Boolean gates, which makes them unsuitable for this work. We used two main ASIC synthesis tools interchangeably: Synopsys Design Compiler (DC) [Syn10] which is one of the most popular commercial logic synthesis tools, and the open-source academic Yosys-ABC toolchain [Wol, Ber]. In the following, we focus on briefly describing the synthesis flow of Synopsys DC.

Synthesis Flow. A Hardware Description Language (HDL) description of the desired circuit is provided to Synopsys DC. Operations in this description get mapped to the most appropriate circuit components selected by Synopsys DC from two types of libraries: the generic technology (GTECH) library of basic logic gates and flip-flops called cells, and *synthetic libraries* consisting of optimized circuit descriptions for more complex operations. Designware [Syn15] is a built-in *synthetic library* provided by Synopsys, consisting of tested IP constructions of standard and complex cells frequently used, such as arithmetic or signal processing operations. This first mapping step is independent of the actual circuit manufacturing technology and results in a generic structural representation of the circuit. This gets mapped next to low-level gates selected from a target *technology library* to obtain a technology-specific representation: a list of Boolean and technology-specific gates (e.g., multiplexers), called netlist.

Synopsys DC performs all of the above mapping and synthesis processes under synthesis and optimization constraints, which are directives and options provided by the developer to optimize the delay, area and other performance metrics of a synthesized circuit.

Input to these hardware synthesis tools can be a pure combinational circuit, which maps only to Boolean gates, or a sequential circuit that requires a clock signal and FF gates which are memory elements to store the current state of the circuit. The output of a sequential circuit is a function of both the circuit inputs and the current state. In this work, we constrain circuit description to combinational circuits.

High-Level Synthesis. Logic synthesis tools accept the input function description most commonly in a HDL format (Verilog or VHDL), whereas more recent logic synthesis tools support high-level synthesis (HLS). This allows them to accept higher-level circuit descriptions in C/C++ or similar high-level programming alternatives. The HLS tools then transform the functional high-level input code into an equivalent hardware circuit description, which in turn can be synthesized by classic logic synthesis. Although this higher abstraction is more developer-friendly and usable, performance of resulting circuits is often inferior to HDL descriptions, unless heavy design constraints are provided to guide the mapping and optimization process.

2.3 The IEEE 754 Floating-Point Standard

Floating-point (FP) numbers allow to represent approximations of real numbers with a trade-off between precision and range. The IEEE 754 floating-point standard [FP008] defines arithmetic formats for finite numbers including signed zeros and subnormal numbers, infinities, and special "Not a Number" values (`NaN`) and rounding rules to be satisfied when rounding numbers during floating-point operations, e.g., rounding to nearest even. Additionally, the standard defines exception handling such as division by zero, overflow, underflow, infinity, invalid and inexact.

The IEEE 754 Standard 32-bit single precision floating-point format consists of 23 bits for significand, 1 bit for sign and 8 bits for exponent distributed from MSB to LSB as follows: sign `[31]`, exponent `[30:23]`, and significand `[22:0]`. The 64-bit double precision format consists of 52 bits for significand, one bit for sign, and 11 bits for exponent.

3. OUR TOOLCHAIN

We describe our toolchain here by presenting our architecture followed by a detailed description of each component.

3.1 Architecture

An overview of our architecture is shown in Fig. 1. We provided the hardware synthesis tools with optimization and synthesis constraints along with a set of customized technology and synthesis libraries (cf. §3.2), to map the input circuit description in Verilog (or any other HDL) into a functionally-equivalent Boolean circuit netlist in Verilog. The output netlist, in the meantime, is constrained to consist of AND, XOR, INV and MUX gates.

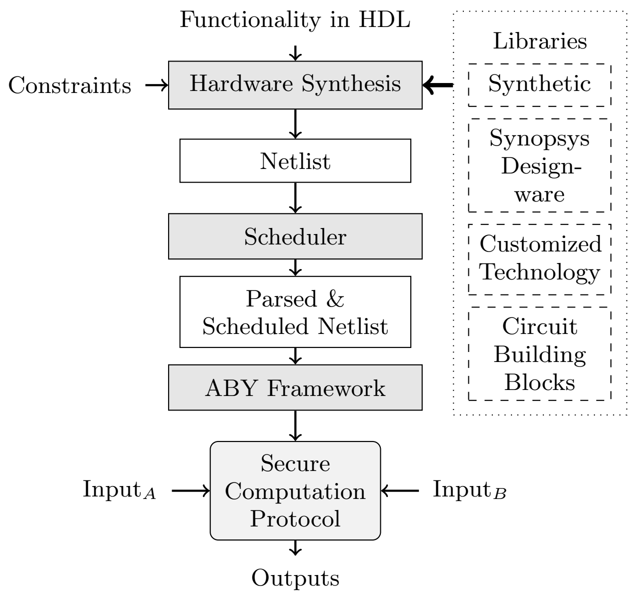


Figure 1: Architecture Overview

The Verilog netlist is then parsed and scheduled, and provided as input to the ABY framework [DSZ15], which we extended to process this netlist and generate the Boolean circuit described in it. The evaluation of the GMW protocol in ABY minimizes the number of communication rounds, i.e., all AND gates on the same layer are evaluated in parallel.

In the following we describe in further detail the main components of our toolchain architecture: logic synthesis (§3.2), scheduling (§3.3), and extending the ABY framework (§3.4).

3.2 Hardware and Logic Synthesis

The GMW protocol and Yao's protocol require that the function to be computed is represented as a Boolean circuit. As described in detail in §1.2, previous work, such as the Fairplay framework [MNPS04, BNP08], used domain-specific high-level languages that allow a developer to describe the function to be computed, which in turn gets compiled into a Boolean circuit. Other compilers allow compilation of circuit descriptions written in C into size-optimized Boolean circuits, e.g., [HFKV12], whereas further tools allow a developer to build up the circuit by instantiating its building blocks from within custom libraries composed of these building blocks, e.g., [HEKM11, Mal11]. All these works rely on custom-made compilers and/or languages which have to compile from a high-level description of the functionality and map it to a Boolean circuit. This may be considered as "reinventing the wheel" since Boolean mapping and optimization is the core of hardware synthesis tools, and has been researched for long. It has been argued, however, that such "hardware compilers" target primarily hardware platforms and therefore involve technology constraints and metrics which are not directly related to the purpose of generating Boolean circuits for secure computation. Writing circuits in HDL, such as Verilog or VHDL, is not entirely high-level, and involves hardware description paradigms which may not be similar to high-level programming paradigms. Furthermore, they rely on the use of sequential logic rather than pure combinational logic.

Exploiting Logic Synthesis. However, the TinyGarble framework [SHS$^+$15] exploited these very same points, and employed hardware synthesis tools in generating compact sequential Boolean circuits for secure evaluation by Yao's garbled circuits protocol [Yao86]. The work in our paper extends this further by using the hardware synthesis tools to generate combinational circuits of more complex functionalities for evaluation by both Yao and the GMW protocol [GMW87], while excluding all design and technology optimization metrics. The synthesis and generation of the Boolean netlist by the synthesis tools (cf. §2.2) can be optimized according to the synthesis constraints and optimization options provided. Hardware synthesis tools conventionally target circuit synthesis on hardware platforms, but can be adapted and exploited for secure computation purposes to generate Boolean netlists which are AND-minimized (depth-optimized primarily for GMW or size-optimized for Yao's garbled circuits).

3.2.1 *Customizing Synthesis*

In the following, we focus on how we customized the synthesis flow of Synopsys DC to generate our Boolean netlists.

Synthesis Flow. The synthesis and optimization constraints that can be provided to Synopsys DC allow us to manipulate it to serve our purposes in this work, and generate depth-optimized circuit netlists for evaluation with GMW. Moreover, we developed a synthetic library of optimized basic cells and depth/size-optimized circuit building blocks that can be assembled by developers to build more complex circuits, and a customized technology library to constrain circuit mapping to XOR and AND gates only. The different libraries and our engineered customizations to achieve this are described next.

Synthetic Libraries. The first step of the synthesis flow is to convert arithmetic and conditional operations (`if-else`, `switch-case`) to their functionally-equivalent logical representations. By default, they are mapped to cells (either simple gates or more complex circuits such as adders and comparators) extracted from the GTECH library and the built-in Synopsys DC DesignWare library [Syn15] (cf. §2.2). A single cell can have different implementations from which the synthesis tool selects, depending on the provided constraints. For example, the sum of two ℓ-bit numbers can be replaced with 1 out of 10 different adder implementations available in both libraries, depending on the optimization constraints provided (optimizing for area or delay).

Our Optimized Circuit Building Blocks Library. Besides the standard built-in libraries, we developed our own DesignWare circuits in a customized *synthetic library*. It consists of depth-optimized circuit descriptions (arithmetic, comparators, 2-to-1 multiplexer, etc.) customized for GMW, as well as size-optimized counterparts for Yao's garbled circuits. Synopsys DC can then be instructed to prefer automated mapping to our customized circuit descriptions (cf. §4) rather than built-in circuits (cf. §3.2.3 for developer usage).

Technology Library. The intermediate generic representation of the circuit obtained in the step before is then mapped into low-level gates extracted from a technology library. A technology library is a library that specifies the gates and cells that can be manufactured by the semiconductor vendor onto the target platform. The library consists of the functional description (such as the Boolean function they represent) of each cell, as well as their performance and technology attributes such as timing parameters (intrinsic rise and fall times, capacitance values, etc.) and area parameters.

Technology libraries targeting ASICs contain a range of cells ranging from simple 2-input gates to more complex gates such as multiplexers and flip-flops. A single cell can also have different implementations which have varying technology attributes. Ultimately, the goal of the synthesis tool is to map the generic circuit description into a generated netlist of cells from this target technology such that user-provided constraints and optimization goals are satisfied.

Our Customized Technology Library. In order to meet our requirements of the Boolean circuit netlists required in this work, we constrain Boolean mapping to non-free AND and free XOR gates. However, Synopsys DC requires that synthesis runs with at least OR, AND and inverter (INV) gates defined in the technology library. We developed a customized technology library which has no manufacturing or technology rules defined, similar to the approach in TinyGarble, and we manipulated the cost functions of the gates by setting the area and delay parameters of XOR gates to 0, and set them to very high non-zero values for OR gates to ensure their exclusion in mapping. Their very high area and delay costs force Synopsys DC to re-map all instances of OR gates to AND and INV gates according to their equivalent Boolean representation ($\mathsf{A} \vee \mathsf{B} = \neg(\neg\mathsf{A} \wedge \neg\mathsf{B})$), and to optimize the Boolean mapping in order to meet the specified area/delay constraints. We set the area and delay costs of an inverter (INV) gate to zero, as they can be replaced with XOR gates with one input buffered to constant one. For AND gates, the area and delay costs are set to reasonably high values, but not too high so that they are not excluded from synthesis. We set MUX gates to area cost equivalent to that of a single AND gate (since the 2-to-1 multiplexer construction in [KS08] is composed of a single AND gate and 2 XOR gates). And we set its delay cost equivalent to 0.25 times more than that of an AND gate to ensure preferred but also non-redundant mapping to MUX gates whenever feasible. We concluded that these settings give the most desirable mapping results after experimenting with Synopsys DC mapping behavior in different scenarios.

Synthesis Constraints. We provide constraints that make delay optimization of the circuit a primary objective followed by area optimization as a secondary objective when generating depth-optimized circuits for GMW. We set the preference attribute to XOR gates, and disable circuit flattening to avoid remapping of XOR gates to other gates. Synthesis tools are not primarily designed to minimize Boolean logic by maximizing XOR gates and reducing the multiplicative complexity of circuits within multi-level logic minimization. This is because XOR gates are only considered as "free" gates in secure computation applications, whereas in the domain of traditional hardware CMOS design, NAND gates are the universal logic gates from which all other gates can be constructed. Hence, the tools need to be heavily manipulated to achieve our objectives. These constraints and technology library settings also have to be customized differently when we want to generate circuits optimized for other secure computation protocols, such as Yao's garbled circuits.

Construction of More Complex Circuits. The customized circuit descriptions we developed can be used to build higher-level and more complex applications. We assembled complex constructions such as Private Set Intersection (PSI) primitives (bitwise-AND, pairwise comparison, and Sort-Compare-Shuffle networks as described in [HEK12]) using our customized building blocks, and they have demonstrated equivalent AND gate count and depth as their hand-optimized counterparts in [HEK12]. In general, all sorts of more complex functionalities and primitives can be constructed by assembling these circuit building blocks along with built-in Designware IP implementations. Consequently, these more complex circuits can then be appended to our library to be re-used in building further more complex circuits, and so on, in a modular and hierarchical way.

HDLs also allow a developer to describe circuits recursively which can be synthesized, which is often the most efficient paradigm for describing depth-optimized circuit constructions such as the depth-optimized "greater than" operation [GSV07], the Waksman permutation network [Wak68], or the Boyar-Peralta counter [BP06].

3.2.2 High-level Function and Operator Mapping

An alternative to describing the circuits for HLS in high-level C/C++ is to allow developers to input their circuit descriptions in high-level Verilog, by calling operators and functions, which we map to "instantiate" circuit modules such as depth-optimized adders or comparators from our customized *synthetic library*. This allows high-level circuit descriptions without incurring the drawbacks of using HLS tools, such as inferior hardware implementation (cf. §2.2).

Mapping operators. We prepared a library description which links our customized circuits into the Synopsys DC. This provides a description of each circuit module, its different implementations, and the operator bound to each module. These operators can be newly created, or already built-in, such as ('+', '-', '*', etc.), but bound to our customized circuits. For instance, when synthesizing the statement `Z = X + Y`, Synopsys DC is automated to map the '+' to our customized Ladner-Fischer adder, rather than a built-in adder implementation.

Mapping Functions. We mapped functions to instantiate circuit modules by creating a global Verilog package file which declares these functions and which circuit modules they instantiate when being called. This package file is then included in the high-level Verilog description code which calls on these functions.

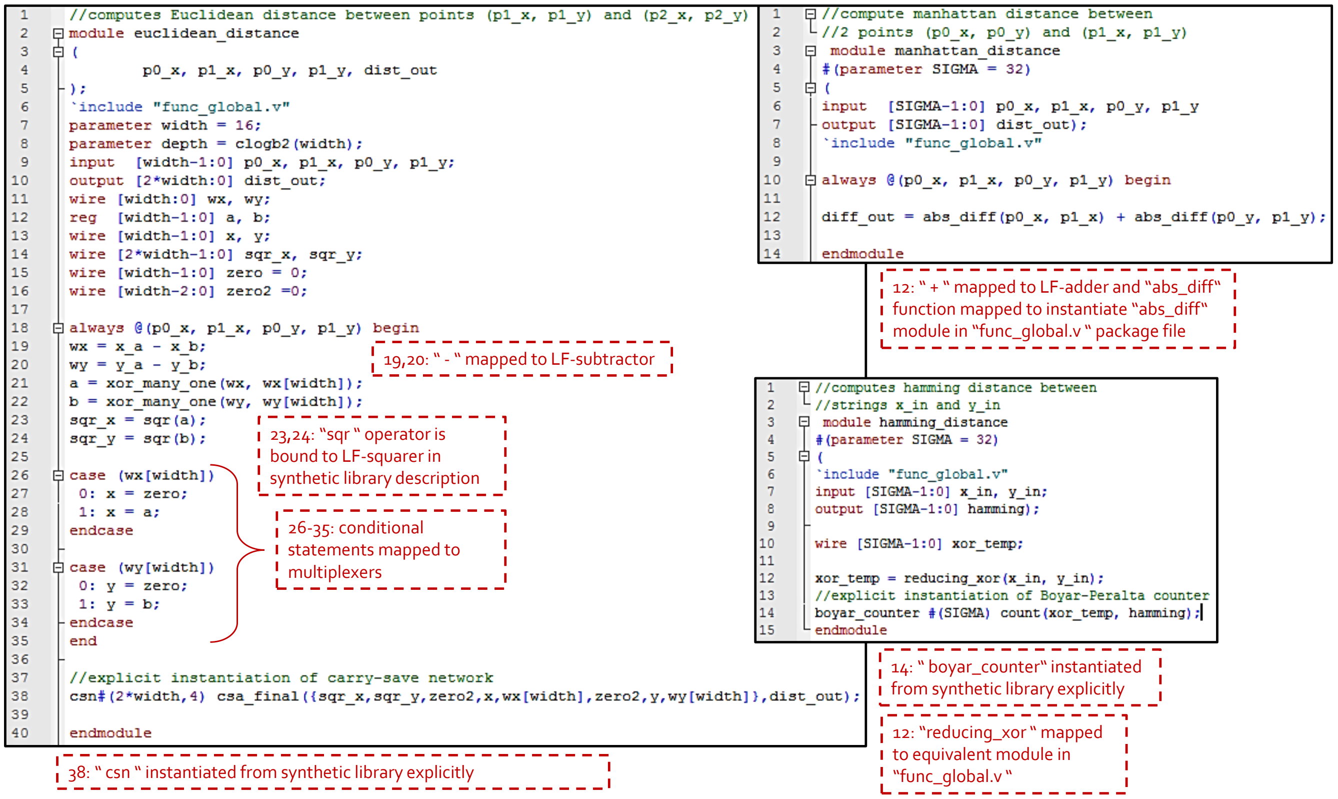
Explicit Instantiation. Other more complex circuits can only be explicitly called from our customized building blocks library, as well as from the Designware IP library which offers a wide range of IP implementations, all of which have verified and guaranteed correctness, such as the floating-point operations we present and benchmark in §5.3. A list of available Designware IP implementations can be found in [Syn15].

High-level Circuit Description Example. In Fig. 2, we show how the depth-optimized constructions of the Manhattan, Euclidean and Hamming distances [SZ13] are described using high-level Verilog. The Manhattan distance between two points is the distance in a 2-dimensional space between these two points based only on horizontal and vertical paths. The Euclidean distance between two points computes the length of the line segment connecting them. Hamming distance between two strings computes the number of positions at which the strings are different.

In the Euclidean distance description, in lines 19 and 20 the '`-`' operator is mapped automatically to our Ladner-Fischer subtractor. The function `sqr` called in lines 23 and 24, is automatically mapped to instantiate our Ladner-Fischer squarer. We declared and bound this function correctly in the package file '`func_global.v`' which is included in line 6. `case` statements (as are `if...else` statements) in lines 26-34 are also mapped to our depth-optimized multiplexer. In line 38, a carry-save network is explicitly instantiated from our library described in §4.2, since some circuit blocks are not mapped to functions and operators and have to be explicitly instantiated due to their structure and design. In the Manhattan distance description, the absolute differences are computed by calling the '`abs_diff`' function in line 12 which is also mapped to instantiate the corresponding circuit. The same high-level abstraction can be seen in the Hamming distance description. Once these distance circuits are constructed, they can be appended to our blocks library to be easily re-used in more complex functionalities.

3.2.3 Developer Usage

By default, Synopsys DC maps operations to Designware circuit descriptions. For operations that have multiple circuit descriptions which are optimized for different parameters, e.g., area or delay, Synopsys DC selects the most appropriate circuit description which best satisfies the constraints provided by the developer in the synthesis script. Alternatively, the developer can explicitly select a specific circuit description to map an operation to. For example, the built-in Designware adder circuit is available in different implementations: ripple-carry, carry-look-ahead and other area- and delay-optimized implementations. Synopsys DC selects the most suitable implementation to map '+' to, depending on the developer-provided constraints. Furthermore, the developer can also specify in the synthesis script that a certain



```
1   //computes Euclidean distance between points (p1_x, p1_y) and (p2_x, p2_y)
2   module euclidean_distance
3   (
4           p0_x, p1_x, p0_y, p1_y, dist_out
5   );
6   `include "func_global.v"
7   parameter width = 16;
8   parameter depth = clogb2(width);
9   input  [width-1:0] p0_x, p1_x, p0_y, p1_y;
10  output [2*width:0] dist_out;
11  wire [width:0] wx, wy;
12  reg  [width-1:0] a, b;
13  wire [width-1:0] x, y;
14  wire [2*width-1:0] sqr_x, sqr_y;
15  wire [width-1:0] zero = 0;
16  wire [width-2:0] zero2 =0;
17
18  always @(p0_x, p1_x, p0_y, p1_y) begin
19  wx = x_a - x_b;
20  wy = y_a - y_b;
21  a = xor_many_one(wx, wx[width]);
22  b = xor_many_one(wy, wy[width]);
23  sqr_x = sqr(a);
24  sqr_y = sqr(b);
25
26  case (wx[width])
27   0: x = zero;
28   1: x = a;
29  endcase
30
31  case (wy[width])
32   0: y = zero;
33   1: y = b;
34  endcase
35  end
36
37  //explicit instantiation of carry-save network
38  csn#(2*width,4) csa_final({sqr_x,sqr_y,zero2,x,wx[width],zero2,y,wy[width]},dist_out);
39
40  endmodule
```

19,20: " - " mapped to LF-subtractor

23,24: "sqr " operator is bound to LF-squarer in synthetic library description

26-35: conditional statements mapped to multiplexers

38: " csn " instantiated from synthetic library explicitly

```
1   //compute manhattan distance between
2   //2 points (p0_x, p0_y) and (p1_x, p1_y)
3    module manhattan_distance
4   #(parameter SIGMA = 32)
5   (
6   input  [SIGMA-1:0] p0_x, p1_x, p0_y, p1_y
7   output [SIGMA-1:0] dist_out);
8   `include "func_global.v"
9
10  always @(p0_x, p1_x, p0_y, p1_y) begin
11
12  diff_out = abs_diff(p0_x, p1_x) + abs_diff(p0_y, p1_y);
13
14  endmodule
```

12: " + " mapped to LF-adder and "abs_diff" function mapped to instantiate "abs_diff" module in "func_global.v " package file

```
1   //computes hamming distance between
2   //strings x_in and y_in
3    module hamming_distance
4   #(parameter SIGMA = 32)
5   (
6   `include "func_global.v"
7   input [SIGMA-1:0] x_in, y_in;
8   output [SIGMA-1:0] hamming);
9
10  wire [SIGMA-1:0] xor_temp;
11
12  xor_temp = reducing_xor(x_in, y_in);
13  //explicit instantiation of Boyar-Peralta counter
14  boyar_counter #(SIGMA) count(xor_temp, hamming);
15  endmodule
```

14: " boyar_counter" instantiated from synthetic library explicitly

12: "reducing_xor " mapped to equivalent module in "func_global.v "

Figure 2: High-level description of the Hamming, Euclidean and Manhattan distances.

implementation is preferred, or the implementation can be explicitly called in the Verilog code.

In order for developers to use our synthetic libraries instead of Designware to map to our customized circuits, they have to decide for which metric to optimize: depth or size. Accordingly, developers add the libraries' paths and a single command in the synthesis script to direct Synopsys DC to optimize for either depth (for GMW) or size (for Yao), and to prefer mapping to which set of circuit descriptions. If developers want to instantiate a specific circuit description from our customized libraries, they can call it by the name of the circuit module and defining its input/output and parameters.

Optimization constraints are generally specified by the developer once for the entire top-level circuit description in the synthesis script, while some sub-circuits require specific optimization constraints. We already specified the optimization constraints for our customized circuit building blocks.

3.2.4 *Challenges of Logic Synthesis for Secure Computation*

Conventionally synthesis tools are best at synthesizing sequential hardware circuits with a clock input and flip-flops. This also means that the actual circuit netlists synthesized are much more compact than combinational Boolean circuits. However, for the purpose of this work, the netlists required are combinational to be evaluated with a secure computation protocol in the ABY framework. This implies synthesis of circuits which reach up to 10 million gates and beyond, which is time- and resource-consuming for hardware synthesis tools. In the hardware synthesis world, this can be managed by generating sub-blocks in a hierarchical fashion, and appending them into one top-level circuit.

However, in this work, one coherent Boolean netlist is required for a single functionality, hence all sub-blocks of a hierarchy must be un-grouped during synthesis, which is resource consuming. We use workarounds to ease the memory and resource requirements. However, this may come at the expense of inter-block optimization across block boundaries, but this can also be customized for individual synthesis scenarios by enabling the boundary optimization option when desired.

3.3 Scheduling

The output netlist generated from the hardware synthesis tools has to be parsed in an intermediate step before being provided to the ABY framework. A parser and scheduler topologically sorts and schedules the netlist gates [KA99], since the Verilog netlist output from some synthesis tools is not topologically sorted, i.e., a wire can be listed as input to one gate before assigning output to it from another. The scheduler generates a Boolean netlist in a format which is similar to Fairplay's SHDL [MNPS04]. All gates and wires are renamed to integer wire IDs for easier processing by the ABY framework, and complex statements are rewritten as one or several available gates. These steps ensure that the final netlist contains only AND, XOR, INV and MUX gates.

3.4 Extending the ABY Framework

The open-source ABY framework [DSZ15] is an extensive tool that enables a developer to manually implement secure two-party computation protocols by offering several low-level as well as intermediate circuit building blocks that can be freely combined. We extended the ABY framework with an interface where externally constructed blocks made of low-level gates can be input in a simple text format, similar to SHDL [MNPS04] and the circuit format from [ST], that we can parse as well, with some modifications.

This interface is used to input the parsed and scheduled netlists from our hardware synthesis. ABY creates a Boolean circuit with low depth from that input netlist, i.e. it schedules AND gates on the earliest possible layer and automatically processes all AND gates in one layer in parallel. A developer has two options: 1) our hardware synthesized netlist can be used as a full protocol instance from private inputs to output or 2) the netlist's functionality can be used as a building block and combined with other synthesized or hand-built sub-circuits within ABY in order to create the whole secure computation protocol. The output of ABY is a fully functional secure computation protocol that is split into setup phase and online phase, that can be evaluated on two parties' private inputs.

4. BULIDING BLOCKS LIBRARY

We implemented the following blocks in Verilog as pure combinational circuits and synthesized their Boolean netlists using both Synopsys DC and Yosys-ABC interchangeably to show that the framework is independent of the used synthesis tool. All implemented circuits have configurable parameters such that they can handle the desired bit-width ℓ of the inputs and/or number of inputs n. We summarize and compare our synthesis results with their hand-optimized counterparts in [HKS$^+$10, HEK12, SZ13]. The two main comparison metrics are size **S** which is the circuit size in terms of non-free AND gates, and depth **D** which is the number of AND gates along the critical path of the circuit. XOR gates are considered to be free, as the GMW protocol and Yao's protocol with free XORs [KS08] allow to securely evaluate XOR gates locally without any communication. Next we show the results for functionalities that have improved depth or size compared with their hand-optimized counterparts in §4.1, and then in §4.2 we describe further functionalities and blocks that we have implemented in our library which show equivalent results as their hand-optimized counterparts. Finally, in §4.3, we describe the floating-point operations and integer division that we benchmark in §5.

4.1 Improved Functionalities

In this section, we present the implemented functionalities that achieved better results in terms of size or depth compared with [HKS$^+$10, SZ13]. Results are given in Tab. 1.

Ladner-Fischer LF Adder/Subtractor. The LF adder/subtractor has a logarithmic depth [LF80, SZ13]. Our results show improvement for both depth (up to 10%) and size (up to 14%) in the subtraction circuit, while maintaining the same size and depth for addition of power-of-two numbers. Both circuits can also handle numbers that are not powers-of-two and achieve better size (up to 20%) as the hardware synthesis tool automatically removes gates whose outputs are neither used later as inputs to other gates nor assigned directly to the output of the circuit.

Karatsuba Multiplier KMUL. We implemented a recursive Karatsuba multiplier [KO62] using a ripple-carry multiplier for inputs with bit-width $\ell < 20$, while for $\ell \geq 20$ inputs are processed recursively. We compare our results with numbers given in [HKS$^+$10], which generated size-optimized Boolean circuits for garbled circuits, but did not consider circuit depth. Here we achieve up to 3% improvement in size.

Manhattan Distance $\mathbf{DST_M}$. Manhattan distance is implemented as a depth-optimized circuit using Ladner-Fischer addition ADD_{LF} and subtraction SUB_{LF} or using ripple-carry addition ADD_{RC} and subtraction SUB_{RC} for a size-optimized circuit [CHK$^+$12, SZ13]. Our results demonstrate improvements in terms of size (up to 16%) and depth (up to 13.6%).

4.2 Further Functionalities

We list further functionalities that we implemented next. Their circuit sizes and depths are equivalent to the hand-optimized circuits in [HEK12, SZ13]: ripple-carry adder and subtractor [BPP00, KSS09], $n \times \ell$-bit carry-save and ripple-carry network adders [Sav97, SZ13], multipliers and squarers [Sav97, KSS09, SZ13], depth-optimized multiplexer [KS08], comparators (equal and greater than) [SZ13], full-adder [SZ13] and Boyar-Peralta counters [BP06, SZ13], and the Sort-Compare-Shuffle circuit for private set intersection (PSI) [HEK12] and its building blocks (bitonic sorter, duplicate-finding circuit, and Waksman permutation network [Wak68]).

Matrix Multiplication. We implemented a size-optimized matrix multiplication circuit that computes one entry in the resulting matrix by computing dot products. This circuit is evaluated such that it computes the entries of the resulting matrix in parallel. Thereby, we can exploit the capability of the ABY framework to evaluate circuits in parallel, which reduces the memory footprint of the implementation. The circuit uses the Karatsuba multiplier and a ripple-carry network adder. It is configurable, i.e., we can set the bit-width ℓ and the number of elements per row or column n. The depths and sizes of these circuits are given in Tab. 3 and their performance is evaluated in §5.2.

4.3 Floating-Point Operations and Integer Division

We generated floating-point operations using the DesignWare library [Syn15], which is a set of building block IPs used to implement, among other operations, floating-point computational circuits for high-end ASICs. The library offers a suite of arithmetic and trigonometric operations, format conversions (integer to floating-point and vice versa) and comparison functions. The provided functionalities are parametrized allowing the developer to select the precision based on either IEEE single or double precision or set a custom-precision format. We can also enable the `ieee_compliance` parameter when we need to guarantee IEEE compatible floating-point numbers ("Not a Number" `NaN` and denormalized numbers). Some functionalities provide an `arch` parameter which can be set for either depth-optimized or size-optimized circuits.

Some of the floating-point functions provide a 3-bit optional input `round`, to determine how the significand should be rounded, e.g. `000` rounds to the nearest even significand which is the IEEE default. They also have an 8-bit optional output flag `status`, in which bits indicate different exceptions of the performed operation allowing error detection. We can choose to truncate or use these `status` bits as desired.

We generated circuits for floating-point addition, subtraction, squaring, multiplication, division, square root, sine, cosine, comparison, exponentiation to base e, exponentiation to base 2, natural logarithm (ln), and logarithm to base 2 for single precision, double precision and a custom 42-bit precision format for comparison with [ABZS13]. The 42-bit format consists of 32 bits for significand, one bit for sign and 9 bits for exponent distributed from MSB to LSB as follows: sign `[41]`, exponent `[40:32]` and significand `[31:0]`. We extended the ABY framework with these floating-point operations and benchmarked them. We give runtimes, depths and sizes for various floating-point operations in §5.3.

We also generated circuits for integer division for different bit-widths $\ell \in \{8, 16, 32, 64\}$ using the built-in DesignWare library [Syn15]. Another possibility for generating division circuits is to use the division operator `'/'` which will be implicitly mapped to the built-in division module in that library. As we optimize for depth our circuits have size $\mathcal{O}(\ell^2 \log \ell) \approx 24\,576$ gates for $\ell = 64$ but low depth 512. In contrast, optimizing for size would yield better size $\mathcal{O}(\ell^2) \approx 3\ell^2 = 12\,288$ gates (for $\mathsf{ADD}/\mathsf{SUB}$, CMP, and MUX), but worse depth $\mathcal{O}(\ell^2) = 4\,096$. We give circuit sizes and depths for integer division in Tab. 2 and benchmarks in §5.1.

5. BENCHMARKS AND EVALUATION

We extended the ABY framework [DSZ15] to read in the parsed and scheduled netlist generated by our hardware synthesis tool and evaluate it with ABY's optimized implementations of the GMW protocol and Yao's garbled circuits (cf. §3.4). In contrast to TinyGarble [SHS$^+$15], which mainly focused on a memory-efficient representation of the circuits and gave only a single example for the time to securely evaluate the circuit, we measure the total execution times for several operations and applications: integer division (§5.1), matrix multiplication (§5.2) and an extensive set of floating-point operations (§5.3). For Yao's protocol we use today's most efficient garbling schemes implemented in the ABY framework [DSZ15]: free XOR [KS08], fixed-key AES garbling with the AES-NI instruction set [BHKR13] and half-gates [ZRE15]. For better comparability of the runtimes we use depth-optimized circuits for both, GMW and Yao.

Compilation and synthesis times for the largest circuits ($\mathsf{FP}_{\mathsf{EXP2}}$, $\mathsf{FP}_{\mathsf{DIV}}$) using Synopsys DC are under 1 hour on a standard PC, but this is only a one-time expense, after which the generated netlist can be re-used without incurring compilation costs again.

We provide runtimes for the *setup phase*, which can be pre-computed independently of the private inputs of the participants and the *online phase*, which takes place after the setup-phase is done and the inputs to the circuit are supplied by both parties. All runtimes are median values of 10 protocol runs. We measured runtimes on two desktop computers with an Intel Core i7 CPU (3.5 GHz) and 16 GB RAM connected via Gigabit-LAN. In all our experiments we set the symmetric security parameter to 128 bits.

5.1 Benchmarks for Integer Division

A complex operation that is not trivially implementable by hand is integer division, as described in §4.3. In Tab. 2 we list the runtime, split in pre-computation phase and online phase and list the circuit parameters for multiple input sizes. We compare our runtime with the runtime prediction of 32-bit integer long division of [KSS13] which we speed up by a factor of 32 and even more for Single Instruction Multiple Data (SIMD) evaluation. We also compare with the runtime of 3-party 64-bit integer division of [ABZS13], which outperforms our single evaluation with GMW by a factor of 1.8. However, for parallel SIMD evaluation we improve upon their runtime by up to factor 3.7. When comparing to the 3-party 32-bit integer division of [BNTW12], we achieve a speedup of 6.5 for single execution, while we require more than 5 times the runtime for 10 000 parallel executions.

5.2 Benchmarks for Matrix Multiplication

Matrix multiplication of integer values is an important use case in many applications. Here we exploit ABY's ability to evaluate circuits in parallel in a SIMD fashion and instantiate dot product computation blocks, each of which calculates a single entry in the result matrix. In Tab. 3 we give the runtimes for dot product computations of 16 values of 16 bit each or 32 values of 32 bit each, as described in §4.2. We compare with the 3-party secret-sharing based implementations of [BNTW12, ZSB13] as well as the 2-party arithmetic-sharing implementation of the ABY framework [DSZ15]. For this comparison we use the values reported in the respective papers and interpolate them to our parameters.

The secret-sharing or artihmetic-sharing based solutions outperform our Boolean Circuits by several orders of magnitude due to their much faster methods for multiplication.

5.3 Benchmarks for Floating-Point Operations

There is a multitude of use cases for floating-point operations in academia and industry, ranging from signal processing to data mining, but due to the complexity of the format it has only recently been considered as application for secure computation [FK11]. Until today there are only few actual implementations of floating-point arithmetic in secure computation, all of which use custom-built protocols [ABZS13, KW14]. Instead, we use multiple standard floating-point building blocks offered by Synopsys DC and synthesize them automatically (cf. §4.3). Tab. 4 depicts the runtime in ms per single floating-point operation, when run once or multiple times in parallel using a SIMD approach. We compare our results for Yao and GMW with hand-optimized floating-point protocols of [ABZS13], who used a 3-party secret sharing approach with security against semi-honest adversaries and desktop computers connected on a Gigabit-LAN for their measurements. The largest runtime improvements can be achieved when evaluating our generated circuits in parallel. We improve the runtime by up to a factor of 21 for parallel evaluation and show similar or somewhat improved runtimes for the lower parallelism levels reported. We can improve upon many results of [KW14] which is in the 3-party setting, except for highly parallel multiplication. We show that our automatically generated circuits are able to outperform hand-crafted circuits in many cases, especially for high degrees of parallelism. We give an application for floating-point arithmetic in §6.

5.4 Benchmark Evaluation

In general, when comparing the implementations of Yao and GMW in the ABY framework, we show that Yao outperforms GMW in most cases but scales much worse, up to a point where the largest circuits cannot be evaluated in parallel, due to the high memory consumption of Yao's protocol. GMW remains beneficial for highly parallel protocol evaluation, as the more critical online time scales almost

Table 1: Synthesis results of improved functionalities compared to hand-optimized circuits for inputs of bit-width ℓ: Ladner-Fischer ADD_{LF}/SUB_{LF}, Karatsuba multiplication KMUL, Manhattan Distance $\mathsf{DST}_{\mathsf{M}}$.

Circuit	**Size S**			**Depth D**		
	Hand-optimized	Ours	Improvement	Hand-optimized	Ours	Improvement
			Depth-Optimized			
ADD_{LF} ($\ell = 20$)	151	121	**20%**	11	11	0%
ADD_{LF} ($\ell = 30$)	226	214	5%	11	11	0%
ADD_{LF} ($\ell = 40$)	361	301	16.6%	13	13	0%
SUB_{LF} ($\ell = 16$)	113	97	**14%**	10	9	10%
SUB_{LF} ($\ell = 32$)	273	241	11%	12	11	8%
SUB_{LF} ($\ell = 64$)	641	577	10%	14	13	7%
DST_{M} ($\ell = 16$)	353	296	**16%**	22	19	**13.6%**
DST_{M} ($\ell = 32$)	825	741	10%	26	23	11.5%
DST_{M} ($\ell = 64$)	1 889	1 778	5.8%	30	27	10%
			Size-Optimized			
KMUL ($\ell = 32$)	1 729	1 697	1.8%	–	63	–
KMUL ($\ell = 64$)	5 683	5 520	2.9%	–	127	–
KMUL ($\ell = 128$)	17 972	17 430	**3%**	–	255	–
DST_{M} ($\ell = 16$)	65	65	0%	34	32	**5.8%**
DST_{M} ($\ell = 32$)	129	129	0%	66	64	3%
DST_{M} ($\ell = 64$)	257	257	0%	130	128	1.5%

Table 2: Runtimes (setup + online phase) in ms per single integer division. '–' indicates that no numbers were given. Protocols marked with * are in the 3-party setting; all other protocols are in the 2-party setting. Entries marked with × could not be run on our machines.

Integer Division	**Parallel Batch Size**			**AND Gates**	
	1	100	10 000	Size	Depth
8-bit GMW	$0.3 + 42.4$	$0.2 + 0.52$	$0.2 + 0.004$	367	32
8-bit Yao	$1.1 + 0.7$	$0.2 + 0.04$	$0.2 + 0.035$	367	32
16-bit GMW	$7.8 + 47.7$	$0.8 + 0.79$	$0.6 + 0.01$	1 542	93
16-bit Yao	$2.0 + 1.1$	$0.7 + 0.14$	$0.7 + 0.14$	1 542	93
32-bit [KSS13]	2 000	–	–	–	–
32-bit [BNTW12]*	400	4	0.5	–	–
32-bit GMW	$3.5 + 58.2$	$3.5 + 3.66$	$2.7 + 0.04$	7 079	207
32-bit Yao	$5.2 + 2.1$	$3.3 + 0.63$	×	7 079	207
64-bit [ABZS13]*	60	41	40	–	–
64-bit GMW	$16.9 + 90.3$	$12.0 + 7.50$	$10.8 + 0.15$	28 364	512
64-bit Yao	$27.5 + 5.6$	$13.1 + 2.49$	×	28 364	512

Table 3: Runtimes (setup + online phase) in ms per single dot product computation, as described in §4.2. Protocols marked with * are in the 3-party setting; all other protocols are in the 2-party setting. Entries marked with × could not be run on our machines. Data from referenced works are interpolated from values given in the respective paper.

Dot Product	**Parallel Batch Size**			**AND Gates**	
	1	100	10 000	Size	Depth
size-optimized RC 16×16-bit GMW	$3.1 + 45.9$	$3.9 + 0.62$	$3.2 + 0.04$	8 427	36
size-optimized RC 16×16-bit Yao	$7.4 + 3.0$	$4.3 + 1.01$	×	8 427	36
32×32-bit Multiplication [BNTW12]*	25.9	0.261	0.058	–	–
32×32-bit Multiplication [ZSB13]*	0.289	0.185	0.184	–	–
32×32-bit Arithmetic Multiplication [DSZ15]	$5.44 + 0.196$	$5.44 + 0.061$	$5.44 + 0.060$	–	–
size-optimized RC 32×32-bit GMW	$55.7 + 68.6$	$21.0 + 1.12$	$21.5 + 0.30$	56 314	69
size-optimized RC 32×32-bit Yao	$76.7 + 18.5$	$28.5 + 6.74$	×	56 314	69

Table 4: Runtimes (setup + online phase) in ms per single floating-point operation for multiple precisions. '–' indicates that no numbers were given. Protocols marked with * are in the 3-party setting; ours are in the 2-party setting. Entries marked with × could not be run on our machines.

FP Operation		Parallel Batch Size					AND Gates	
		1	10	100	1 000	10 000	Size	Depth
	32-bit GMW	0.4 + 39.6	0.1 + 4.1	0.1 + 0.45	0.1 + 0.06	0.1 + 0.003	218	12
	32-bit Yao	1.1 + 0.7	0.3 + 0.1	0.5 + 0.03	0.1 + 0.03	0.1 + 0.033	218	12
	42-bit [ABZS13]*	–	5.4	3.2	2.3	2.2	–	–
$\mathsf{FP}_{\mathsf{CMP}}$	42-bit GMW	0.4 + 39.6	0.2 + 4.3	0.2 + 0.44	0.2 + 0.05	0.1 + 0.003	290	13
	42-bit Yao	1.0 + 0.7	0.3 + 0.1	0.2 + 0.04	0.2 + 0.04	0.2 + 0.043	290	13
	64-bit GMW	0.4 + 40.6	0.3 + 4.3	0.2 + 0.49	0.2 + 0.05	0.2 + 0.004	427	15
	64-bit Yao	1.1 + 0.7	0.3 + 0.1	0.2 + 0.06	0.2 + 0.06	0.2 + 0.065	427	15
	32-bit [KW14]*	1 370	137.0	14.5	1.9	1.6	–	–
	32-bit GMW	3.0 + 46.1	1.1 + 5.3	1.0 + 0.66	0.7 + 0.06	0.7 + 0.01	1 820	59
	32-bit Yao	2.0 + 1.1	1.0 + 0.2	0.9 + 0.17	0.9 + 0.17	0.9 + 0.18	1 820	59
	42-bit [ABZS13]*	–	19.0	11.0	9.3	9.1	–	–
$\mathsf{FP}_{\mathsf{ADD}}$	42-bit GMW	5.3 + 46.3	1.5 + 5.8	1.3 + 1.07	1.0 + 0.07	0.9 + 0.02	2 490	69
	42-bit Yao	2.6 + 1.3	1.3 + 0.3	1.2 + 0.24	1.2 + 0.23	1.2 + 0.24	2 490	69
	64-bit [KW14]*	1 471	147.1	16.7	4.8	4.1	–	–
	64-bit GMW	2.1 + 46.9	2.2 + 6.3	2.3 + 0.73	1.6 + 0.03	1.6 + 0.03	4 303	72
	64-bit Yao	3.6 + 1.6	2.2 + 0.5	2.0 + 0.40	2.0 + 0.40	2.0 + 0.40	4 303	72
	32-bit [KW14]*	434.8	43.5	4.4	0.6	0.2	–	–
	32-bit GMW	1.8 + 42.9	1.6 + 5.6	1.4 + 0.67	1.1 + 0.05	1.1 + 0.02	3 016	47
	32-bit Yao	8.1 + 1.1	1.6 + 0.3	1.4 + 0.27	1.4 + 0.27	1.4 + 0.29	3 016	47
	42-bit [ABZS13]*	–	4.2	3.4	3.2	3.1	–	–
$\mathsf{FP}_{\mathsf{MULT}}$	42-bit GMW	2.0 + 47.3	2.4 + 6.3	2.6 + 0.82	1.9 + 0.08	1.8 + 0.03	4 757	72
	42-bit Yao	4.1 + 1.7	2.5 + 0.5	2.2 + 0.43	2.2 + 0.43	2.2 + 0.43	4 757	72
	64-bit [KW14]*	476.2	47.6	5.1	0.9	0.3	–	–
	64-bit GMW	15.5 + 170.1	5.6 + 8.7	5.0 + 0.95	4.1 + 0.08	4.2 + 0.05	11 068	111
	64-bit Yao	13.3 + 2.7	5.4 + 1.1	5.2 + 1.00	5.1 + 0.99	×	11 068	111
	32-bit [KW14]*	11 111	1 177	142.9	41.7	31.3	–	–
	32-bit GMW	1.3 + 57.7	1.2 + 6.6	1.2 + 1.22	0.9 + 0.12	0.8 + 0.01	2 455	197
	32-bit Yao	2.6 + 0.8	1.5 + 0.3	1.2 + 0.23	1.2 + 0.22	1.2 + 0.23	2 455	197
$\mathsf{FP}_{\mathsf{SQRT}}$	42-bit GMW	2.6 + 66.4	2.2 + 8.8	2.4 + 1.69	1.6 + 0.15	1.6 + 0.03	4 810	300
	42-bit Yao	3.9 + 1.2	2.4 + 0.5	2.3 + 0.43	2.2 + 0.42	2.2 + 0.44	4 810	300
	64-bit [KW14]*	12 500	1 316	217.4	103.1	96.2	–	–
	64-bit GMW	10.5 + 87.4	6.4 + 14.9	5.1 + 6.23	4.3 + 0.23	4.3 + 0.06	12 706	557
	64-bit Yao	9.4 + 2.6	6.2 + 1.3	6.3 + 1.14	5.9 + 1.12	×	12 706	557
	32-bit [KW14]*	6 250	625.0	71.4	16.9	12.7	–	–
	32-bit GMW	2.3 + 64.3	3.1 + 9.3	2.6 + 1.78	2.0 + 0.16	2.0 + 0.03	5 395	296
	32-bit Yao	4.2 + 1.9	2.7 + 0.6	2.5 + 0.49	2.5 + 0.49	2.5 + 0.49	5 395	296
	42-bit [ABZS13]*	–	15.0	12.0	12.0	12.0	–	–
$\mathsf{FP}_{\mathsf{DIV}}$	42-bit GMW	9.9 + 79.8	5.4 + 13.0	4.6 + 2.48	3.7 + 0.23	3.7 + 0.05	9 937	462
	42-bit Yao	7.0 + 2.7	4.9 + 1.0	4.7 + 0.90	4.6 + 0.89	×	9 937	462
	64-bit [KW14]*	6 667	666.7	83.3	43.5	19.2	–	–
	64-bit GMW	16.6 + 123.4	12.5 + 25.4	8.4 + 4.92	8.6 + 0.38	8.7 + 0.12	22 741	994
	64-bit Yao	15.2 + 5.0	11.1 + 2.4	10.6 + 2.06	10.6 + 2.09	×	22 741	994
	32-bit GMW	5.5 + 144.2	5.2 + 14.7	4.7 + 0.85	3.7 + 0.09	3.8 + 0.05	9 740	100
	32-bit Yao	6.5 + 1.8	4.7 + 0.9	4.5 + 0.84	4.5 + 0.83	×	9 740	100
	42-bit [ABZS13]*	–	88.0	80.0	75.0	75.0	–	–
$\mathsf{FP}_{\mathsf{EXP2}}$	42-bit GMW	14.5 + 179.1	12.6 + 23.7	10.2 + 1.14	9.4 + 0.17	9.3 + 0.12	24 357	156
	42-bit Yao	15.8 + 4.4	11.9 + 2.4	11.3 + 2.13	11.2 + 2.14	×	24 357	156
	64-bit GMW	16.7 + 455.1	12.2 + 88.9	9.2 + 17.33	8.1 + 0.51	8.2 + 0.12	21 431	1214
	64-bit Yao	14.3 + 4.2	10.6 + 2.2	10.0 + 1.91	9.9 + 1.89	×	21 431	1214
	32-bit GMW	4.1 + 67.0	5.7 + 8.0	5.0 + 1.48	4.1 + 0.10	4.0 + 0.05	10 568	157
	32-bit Yao	7.0 + 2.1	5.1 + 1.0	4.9 + 0.91	4.9 + 0.90	×	10 568	157
	42-bit [ABZS13]*	–	159.0	103.0	97.0	96.0	–	–
$\mathsf{FP}_{\mathsf{LOG2}}$	42-bit GMW	16.0 + 67.4	12.5 + 20.5	9.8 + 2.80	8.5 + 0.19	8.9 + 0.11	23 041	266
	42-bit Yao	15.9 + 4.1	11.1 + 2.3	10.7 + 2.01	10.6 + 1.99	×	23 041	266
	64-bit GMW	19.7 + 95.8	11.0 + 32.1	8.5 + 6.34	7.6 + 0.45	7.6 + 0.10	19 789	649
	64-bit Yao	13.3 + 3.9	9.7 + 2.0	9.2 + 1.76	9.2 + 1.75	×	19 789	649

linearly with the level of parallelism. The setup times of Yao and GMW are similar for all parameters.

Our improved performance stems from both, the optimized circuits generated by the state-of-the-art hardware synthesis tools which we manipulate to optimize the circuits for either depth or size, and from the efficient implementation of GMW and Yao's garbled circuits with most recent optimizations in ABY. Since both protocols are based on Boolean circuits, we improve the performance of operations that require many bit operations. Operations that involve many integer multiplications are better suited for solutions based on arithmetic- or secret-sharing.

6. APPLICATION: PRIVACY-PRESERVING PROXIMITY TESTING ON EARTH

As application for secure computation on floating-point operations, we consider privacy-preserving proximity testing on Earth [ŠG14]. Here, the goal is to compute if two coordinates C_A and C_B input by party A and B respectively are within a given distance ϵ: $D(C_A, C_B) < \epsilon$. This is a useful but rather privacy-critical use case that has many applications, such as finding nearby friends, points of interest or targeted advertising, and is widely used with the recent spread of end-user GPS receivers and geo location via IP addresses. The authors of [ŠG14] present and compare three different distance metrics: UTM, ECEF, and HS described below. In their paper, the authors design secure protocols based on additively homomorphic encryption (HE) or Yao's garbled circuits (GC) that require to quantize all values to integers, which means a loss of precision. Instead, our framework allows to compute the distance formulas directly on floating-point numbers with multiple precision options available and thus can offer a higher precision.

Universal Transverse Mercator (UTM). This distance metric maps Earth over a set of planes and provides accurate results if A and B are located relatively close to each other, within the same UTM zone.

In this metric coordinates are expressed as 2-dimensional points: $C_A = (x_A, y_A)$ and $C_B = (x_B, y_B)$.

$D_{\text{UTM}}(C_A, C_B) < \epsilon \Leftrightarrow (\underline{x_A} - x_B)^2 + (\underline{y_A} - y_B)^2 < \epsilon^2$, where underlined variables are inputs of party A and the other terms are inputs of party B. For computing this formula we need 2 $\mathsf{FP}_{\mathsf{SQR}}$, 3 $\mathsf{FP}_{\mathsf{ADD}}$, and 1 $\mathsf{FP}_{\mathsf{CMP}}$ operations.

Earth-Centered, Earth-Fixed (ECEF). This distance metric uses the Earth-Centered, Earth-Fixed (ECEF, also known as Earth Centered Rotational, or ECR) coordinate system which provides very accurate results when the parties are far apart.

The coordinates are expressed as 3-dimensional points where $(0, 0, 0)$ is the center of the Earth: $C_A = (x_A, y_A, z_A)$ and $C_B = (x_B, y_B, z_B)$.

$D_{\text{ECEF}}(C_A, C_B) < \epsilon \Leftrightarrow$
$(\underline{x_A} - x_B)^2 + (\underline{y_A} - y_B)^2 + (\underline{z_A} - z_B)^2 < 4R^2 a_\epsilon$,

with $a_\epsilon = \dfrac{(\tan \frac{\epsilon}{2R})^2}{1 + (\tan \frac{\epsilon}{2R})^2}$. Underlined variables are inputs of party A and the other terms are inputs of party B. Computing this formula takes 3 $\mathsf{FP}_{\mathsf{SQR}}$, 5 $\mathsf{FP}_{\mathsf{ADD}}$, and 1 $\mathsf{FP}_{\mathsf{CMP}}$ operations.

Haversine (HS). This distance metric is based on the haversine (HS) formula which is a trigonometric formula used to compute distances on a sphere and is very accurate regardless of the position of A and B.

The coordinates are expressed as spherical coordinates with latitude (lat) and longitude (lon): $C_A = (\text{lat}_A, \text{lon}_A)$ and $C_B = (\text{lat}_B, \text{lon}_B)$.

$D_{\text{HS}}(C_A, C_B) < \epsilon \Leftrightarrow$
$\underline{\alpha^2} \cdot \beta^2 - \underline{2\alpha\gamma} \cdot \beta\delta + \underline{\gamma^2} \cdot \delta^2 + \underline{\zeta\theta^2} \cdot \eta\lambda^2 - \underline{2\zeta\theta\mu} \cdot \eta\lambda\nu + \underline{\zeta\mu^2} \cdot \eta\nu^2 < a_\epsilon$,
with a_ϵ as defined above and

$$\begin{array}{ll}
\alpha = \cos(\text{lat}_A/2) & \beta = \sin(\text{lat}_B/2) \\
\gamma = \sin(\text{lat}_A/2) & \delta = \cos(\text{lat}_B/2) \\
\zeta = \cos(\text{lat}_A) & \eta = \cos(\text{lat}_B) \\
\theta = \sin(\text{lon}_A/2) & \lambda = \cos(\text{lon}_B/2) \\
\mu = \cos(\text{lon}_A/2) & \nu = \sin(\text{lon}_B/2).
\end{array}$$

Underlined terms are inputs of party A while all other terms are inputs of party B. Computing this formula requires 6 $\mathsf{FP}_{\mathsf{MULT}}$, 5 $\mathsf{FP}_{\mathsf{ADD}}$, and 1 $\mathsf{FP}_{\mathsf{CMP}}$ operations.

Performance. We implemented the three proximity testing algorithms from [ŠG14] using our floating-point building blocks. In Tab. 5 we compare the runtime of the original implementation of [ŠG14] that uses homomorphic encryption (HE) and Yao's Garbled Circuits (GC) with our implementation based on GMW and Yao for single and parallel evaluation. We are able to achieve better runtimes for single executions of the protocol (by factor 6.2 for HS and more than factor 14 for UTM and ECEF), and more than two orders of magnitude speedup for highly parallel execution. Thereby, we show that our approach allows to substantially improve upon the runtime of hand-crafted protocols while at the same time it benefits from the heavily tested and verified circuit building blocks from industrial-grade hardware synthesis libraries.

Acknowledgments

We thank the anonymous reviewers of ACM CCS 2015 for their helpful comments. This work was supported by the European Union's 7th Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE), by the DFG as part of project E3 within the CRC 1119 CROSSING, by the German Federal Ministry of Education and Research (BMBF) within the European Center for Security and Privacy by Design (EC SPRIDE), by the Hessian LOEWE excellence initiative within the Center for Advanced Security Research Darmstadt (CASED), and in part by the ONR grant number R17460 and NSF grant R3F530 to Rice University.

Table 5: Runtimes (setup + online phase) in ms per single proximity test for multiple precisions. '–' indicates that no numbers were given. All protocols are in the 2-party setting. Entries marked with $\times$ could not be run on our machines.

Distance Metric		Parallel Batch Size			AND Gates	
		1	100	10 000	Size	Depth
UTM	HE [ŠG14]	700 … 1 100	–	–	–	–
	GC [ŠG14]	401.0 + 102.0	–	–	–	–
	32-bit GMW	4.4 + 59.8	4.0 + 1.49	3.3 + 0.05	8 815	146
	32-bit Yao	18.0 + 2.4	4.2 + 0.87	$\times$	8 815	146
	64-bit GMW	19.9 + 67.2	10.6 + 2.65	10.2 + 0.14	26 588	195
	64-bit Yao	18.1 + 5.7	12.5 + 2.54	$\times$	26 588	195
ECEF	HE [ŠG14]	1 000 … 1 300	–	–	–	–
	GC [ŠG14]	404.0 + 105.0	–	–	–	–
	32-bit GMW	5.7 + 60.1	5.8 + 1.56	5.3 + 0.07	14 042	205
	32-bit Yao	12.8 + 3.3	6.6 + 1.32	$\times$	14 042	205
	64-bit GMW	13.9 + 78.1	15.8 + 2.91	16.0 + 0.20	41 850	267
	64-bit Yao	27.4 + 8.8	19.9 + 3.88	$\times$	41 850	267
HS	HE [ŠG14]	1 700	–	–	–	–
	GC [ŠG14]	409.0 + 124.0	–	–	–	–
	32-bit GMW	13.6 + 67.5	11.6 + 2.11	10.5 + 0.14	27 525	224
	32-bit Yao	17.9 + 5.6	12.8 + 2.48	$\times$	27 525	224
	64-bit GMW	49.5 + 283.6	33.3 + 3.40	33.4 + 0.41	88 530	342
	64-bit Yao	67.8 + 18.0	41.4 + 8.03	$\times$	88 530	342

7. REFERENCES

[ABZS13] M. Aliasgari, M. Blanton, Y. Zhang, A. Steele. Secure computation on floating point numbers. In *NDSS'13*. The Internet Society, 2013.

[ALSZ13] G. Asharov, Y. Lindell, T. Schneider, M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *ACM CCS'13*, p. 535–548. ACM, 2013.

[Bea91] D. Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO'91*, volume 576 of *LNCS*, p. 420–432. Springer, 1991.

[Ber] Berkeley Logic Synthesis. ABC: a system for sequential synthesis and verification, release 70930. http://www.eecs.berkeley.edu/~alanmi/abc/.

[BHKR13] M. Bellare, V. Hoang, S. Keelveedhi, P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE S&P'13*, p. 478–492. IEEE, 2013.

[BNP08] A. Ben-David, N. Nisan, B. Pinkas. FairplayMP: a system for secure multi-party computation. In *ACM CCS'08*, p. 257–266. ACM, 2008.

[BNTW12] D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

[BP06] J. Boyar, R. Peralta. Concrete multiplicative complexity of symmetric functions. In *Mathematical Foundations of Computer Science (MFCS'06)*, volume 4162 of *LNCS*, p. 179–189. Springer, 2006.

[BPP00] J. Boyar, R. Peralta, D. Pochuev. On the multiplicative complexity of boolean functions over the basis $(\wedge, \oplus, 1)$. *Theoretical Computer Science*, 235(1):43–57, 2000.

[CHK$^+$12] S.-G. Choi, K.-W. Hwang, J. Katz, T. Malkin, D. Rubenstein. Secure multi-party computation of Boolean circuits with applications to privacy in on-line marketplaces. In *CT-RSA'12*, volume 7178 of *LNCS*, p. 416–432. Springer, 2012.

[DSZ15] D. Demmler, T. Schneider, M. Zohner. ABY – a framework for efficient mixed-protocol secure two-party computation. In *NDSS'15*. The Internet Society, 2015. Code: https://github.com/encryptogroup/ABY.

[FK11] M. Franz, S. Katzenbeisser. Processing encrypted floating point signals. In *ACM Multimedia and Security (MM&Sec'11)*, p. 103–108. ACM, 2011.

[FP008] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, p. 1–70, Aug 2008.

[GMW87] O. Goldreich, S. Micali, A. Wigderson. How to play any mental game. In *STOC'87*, p. 218–229. ACM, 1987.

[GSV07] J. Garay, B. Schoenmakers, J. Villegas. Practical and secure solutions for integer comparison. In *PKC'07*, volume 4450 of *LNCS*, p. 330–342. Springer, 2007.

[HEK12] Y. Huang, D. Evans, J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS'12*. The Internet Society, 2012.

[HEKM11] Y. Huang, D. Evans, J. Katz, L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security'11*, p. 539–554. USENIX, 2011.

[HFKV12] A. Holzer, M. Franz, S. Katzenbeisser, H. Veith. Secure two-party computations in ANSI C. In *ACM CCS'12*, p. 772–783. ACM, 2012.

[HKS$^+$10] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *ACM CCS'10*, p. 451–462. ACM, 2010.

[IKNP03] Y. Ishai, J. Kilian, K. Nissim, E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO'03*, volume 2729 of *LNCS*, p. 145–161. Springer, 2003.

[KA99] Y.-K. Kwok, I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[KO62] A. A. Karatsuba, Y. Ofman. Multiplication of many-digital numbers by automatic computers. *SSSR Academy of Sciences*, 145:293–294, 1962.

[KS08] V. Kolesnikov, T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP'08*, volume 5126 of *LNCS*, p. 486–498. Springer, 2008.

[KSMB13] B. Kreuter, A. Shelat, B. Mood, K. R. B. Butler. PCF: A portable circuit format for scalable two-party secure computation. In *USENIX Security'13*, p. 321–336. USENIX, 2013.

[KSS09] V. Kolesnikov, A.-R. Sadeghi, T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *CANS'09*, volume 5888 of *LNCS*, p. 1–20. Springer, 2009.

[KSS12] B. Kreuter, A. Shelat, C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security'12*, p. 285–300. USENIX, 2012.

[KSS13] F. Kerschbaum, T. Schneider, A. Schröpfer. Automatic protocol selection in secure two-party computations. In *ACNS'15*, volume 8479 of *LNCS*, p. 1–18. Springer, 2013.

[KW14] L. Kamm, J. Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, p. 1–18, 2014.

[LF80] R. E. Ladner, M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[LWN$^+$15] C. Liu, X. S. Wang, K. Nayak, Y. Huang, E. Shi. ObliVM: A programming framework for secure computation. In *IEEE S&P'15*, p. 359–376. IEEE, 2015.

[Mal11] L. Malka. VMCrypt - modular software architecture for scalable secure computation. In *ACM CCS'11*, p. 715–724. ACM, 2011.

[MLB12] B. Mood, L. Letaw, K. R. B. Butler. Memory-efficient garbled circuit generation for mobile devices. In *FC'12*, volume 7397 of *LNCS*, p. 254–268. Springer, 2012.

[MNPS04] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella. Fairplay – a secure two-party computation system. In *USENIX Security'04*, p. 287–302. USENIX, 2004.

[NNOB12] J. B. Nielsen, P. S. Nordholt, C. Orlandi, S. S. Burra. A new approach to practical active-secure two-party computation. In *CRYPTO'12*, volume 7417 of *LNCS*, p. 681–700. Springer, 2012.

[Sav97] J. E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Pub, Boston, MA, USA, 1st edition, 1997.

[ŠG14] J. Šeděnka, P. Gasti. Privacy-preserving distance computation and proximity testing on earth, done right. In *ACM ASIACCS'14*, p. 99–110. ACM, 2014.

[SHS$^+$15] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, F. Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *IEEE S&P'15*, p. 411–428. IEEE, 2015.

[ST] N. Smart, S. Tillich. Circuits of basic functions suitable for MPC and FHE. http://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/.

[Syn10] Synopsys Inc. Design compiler, 2010. http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler.

[Syn15] Synopsys Inc. DesignWare library - datapath and building block IP. https://www.synopsys.com/dw/buildingblock.php, 2015.

[SZ13] T. Schneider, M. Zohner. GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In *FC'13*, volume 7859 of *LNCS*, p. 275–292. Springer, 2013.

[Wak68] A. Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, 1968.

[Wol] C. Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.

[Yao86] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS'86*, p. 162–167. IEEE, 1986.

[ZRE15] S. Zahur, M. Rosulek, D. Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *EUROCRYPT'15*, volume 9057 of *LNCS*, p. 220–250. Springer, 2015.

[ZSB13] Y. Zhang, A. Steele, M. Blanton. PICCO: a general-purpose compiler for private distributed computation. In *ACM CCS'13*, p. 813–826. ACM, 2013.