2019 IEEE European Symposium on Security and Privacy (EuroS&P)

EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning

Nishanth Chandran*, Divya Gupta*, Aseem Rastogi*, Rahul Sharma*, Shardul Tripathi†
*Microsoft Research, Bangalore, India
{nichandr, divya.gupta, aseemr, rahsha}@microsoft.com
† Indian Institute of Technology, New Delhi, India
shardul.511@gmail.com

***Abstract*—We present EzPC, a secure two-party computation (2PC) framework that generates efficient 2PC protocols from high-level, easy-to-write programs. EzPC provides formal correctness and security guarantees while maintaining performance and scalability. Previous language frameworks, such as CBMC-GC, ObliVM, SMCL, and Wysteria, generate protocols that use either arithmetic or boolean circuits exclusively. Our compiler is the first to generate protocols that combine both arithmetic and boolean circuits for better performance. We empirically demonstrate that the performance of the protocols generated by EzPC is comparable to or better than (in some cases upto 19x) their state-of-the-art, hand-crafted implementations, while EzPC protocols also outperform their boolean circuits only counterparts by as much as 25x.**

I. Introduction

Today it is hard for developers to program secure applications using cryptographic techniques. Typical developers lack a deep understanding of cryptographic protocols, and cannot be expected to use them correctly and efficiently on their own. Ideally, a developer would declare the functionality in a general purpose, high-level programming language and a tool, e.g. a compiler, would generate an efficient protocol that implements the functionality securely, while hiding the cryptography behind-the-scenes.

This paper presents such a framework for Secure Two-party Computation (2PC), a powerful cryptographic technique that allows two mutually distrusting parties to compute a publicly known joint function of their secret inputs in a way that both the parties learn nothing about the inputs of each other beyond what is revealed by their (possibly different) outputs. For example, 2PC can be used for *secure prediction* ([1], [2], [3], [4], [5]), where one party (the server) holds a proprietary classifier to predict a label (e.g., a disease, genomics, or spam detection), and the other party (the client) holds a private input that it wants to run the classifier on. Using 2PC guarantees that the server learns nothing about the client's input or output, and that the client learns nothing about the classifier, beyond what is revealed by the output label.

To understand the state-of-the-art, let us consider an example underlying many secure prediction algorithms. Suppose Alice wants to write a 2PC protocol to securely compute $w^T x > b$. Here w (a vector) and b (a scalar) constitute the server classifier, and x is the client's input vector. Further, $\cdot^T$ is the matrix transpose operator, and $w^T x$ denotes the inner product of w^T and x. Alice has the following options.

She can program the computation in one of the several programmer friendly, domain-specific languages (such as Fairplay [6], Wysteria [7], ObliVM [8], CBMC-GC [9], SMCL [10], Sharemind [11], [12] etc.) that would automatically compile it to a 2PC protocol. However, all of these frameworks use cryptographic backends that take as input the computation expressed either as a boolean circuit ([13], [14]) or as an arithmetic circuit ([15], [16], [17]). The efficiency of the generated 2PC protocol is thus bounded by the efficiency of representing the computation in *one* of these representations. For instance, multiplication of two ℓ-bit integers can either be expressed as a boolean circuit of size $O(\ell^2)$, or as an arithmetic circuit with 1 multiplication gate. It is well-known that boolean circuits are not suitable for doing arithmetic operations such as integer multiplications but are unavoidable for boolean operations such as comparison [18], [19], [20], [5], [4], [21]. For better efficiency, Alice would ideally like to compute $w^T x$ using an arithmetic circuit, and the comparison with b using a boolean circuit. Unfortunately, none of the above frameworks support combinations of arithmetic and boolean circuits, and using different tools for different parts of the computation is cumbersome and error-prone.

Alternatively, Alice can use a tool such as ABY (Demmler et al. [18]) that allows the computation to be expressed as a combination of arithmetic and boolean circuits. However, here, the programming interface is quite low-level: the programmer is required to first manually split the computation into arithmetic and boolean components, and then write the circuits for all the components manually, including the appropriate interconversion gates between them. Although some of this work can be automated [22], [23], [19], [20], [24], writing large computations at a circuit level in such frameworks can be tedious (a sentiment echoed by Demmler et al. [18] themselves).

A third option for Alice is to earn a PhD in cryptography, and design and implement specialized, efficient 2PC protocols (similar to [1], [2], [4], [25]) for her tasks.

A. Our Contributions

This paper presents EzPC[1], the first "cryptographic cost aware" compiler that generates efficient and scalable 2PC

[1]Read as "easy peasy", stands for Easy 2 Party Computation. The implementation is available at https://github.com/mpc-msri/EzPC.

DOI 10.1109/EuroSP.2019.00043

```
uint w[30] = input1();  uint b = input1();
uint x[30] = input2();
uint acc = 0;
for i in [0 : 30] { acc = acc + (w[i] × x[i]); }
output2((acc > b) ? 1 : 0) //only to party 2
```

Fig. 1: EzPC code for $w^T x > b$

protocols using combinations of arithmetic and boolean circuits. EzPC source language is a simple imperative language absent of any cryptographic details. The compiler is backed by a formal model that enables it to choose arithmetic or boolean representations for different parts of the program, while automatically inserting interconversion gates as necessary. In addition to guiding the compiler, the formal model also provides strong correctness and security theorems. Our comprehensive evaluation shows that the automatically generated protocols have performance comparable to or better than the custom, specialized protocols from previous works [1], [2], [4], [5], [26], [25]. In fact, these papers (and others) cite the inefficiency of generic 2PC as the major motivation behind the design of specialized protocols. Using EzPC, we empirically demonstrate that generic 2PC implementations are much more efficient at ML tasks than what they were believed to be. Below we describe the salient features of EzPC.

1) Ease of programming: EzPC source programs are ideal functionalities that describe "what" computation needs to be done, rather than "how" to do it. In particular, the programmer writes the high-level computation without thinking about the underlying cryptographic details. For example, Figure 1 shows an EzPC source program for $w^T x > b$. The program is quite similar to what a programmer might write in C++ or Java. The simplicity of the language comes with the usual benefits: it is easily accessible to the developers, there are fewer avenues for making mistakes, developers don't bear the burden of getting cryptographic details right, code optimizations can be left to the compiler, and it is easy to maintain and modify the programs. Needless to say, frameworks that expose low-level circuit APIs to the programmer do not enjoy these benefits.

2) Cryptographic cost aware compiler: The EzPC compiler compiles a source program to a hybrid computation consisting of *public* and *secret* parts. In the example above, for instance, EzPC compiler realizes that the array index `i` is public, and generates non-cryptographic code for the array accesses. Further, within the secret parts, EzPC compiler is aware of the cryptographic costs of arithmetic and boolean representations of the source language operators. Based on these costs, the compiler automatically picks arithmetic or boolean representations for different sub-parts, and generates the corresponding circuits along with the required interconversion gates. The outcome is an efficient 2PC protocol combining arithmetic and boolean circuits, while the programmer remains oblivious of all these cryptographic details. Indeed, EzPC is the first such cryptographic cost aware compiler.

3) Scalability (secure code pipelining): 2PC tools often do not scale to large functionalities. The reason is that most 2PC implementations use a circuit-like representation as an intermediate language. Hence, the largest compute that can be done securely is upper-bounded by the largest circuit that can fit in the machine memory[2]. This is a show-stopper for applications like secure machine learning, secure prediction, etc. that operate on large data. EzPC addresses the scalability concern using a novel technique that we call secure code pipelining (or SCP in short). At a high level, we decompose the original program into a sequence of small sub-programs, which are then sequentially processed by EzPC, while appropriately threading the intermediate outputs along. While this addresses the scalability concern (i.e., the circuit sizes of the sub-programs are now small enough to fit in the memory), we still have to address the security risk of revealing the intermediate outputs. EzPC comes to the rescue: it automatically inserts the required instrumentation to ensure security of these intermediate outputs (Section V). As we show in our evaluation, SCP allows us to program large applications in EzPC. Furthermore, all prior pipelining approaches work at circuit level and are intimately tied to garbled circuits. SCP operates at source code level and this design ensures compatibility with arbitrary 2PC protocols.

4) Formal guarantees: We prove formal correctness and security theorems for our compiler. The correctness theorem relates the "trusted third party" semantics of a source program and the "protocol" semantics (the distributed 2PC semantics that relies on circuit evaluation) of the corresponding compiled program. The theorem guarantees that for all well-typed source programs, the two semantics successfully terminate (e.g., there are no array index out-of-bounds errors) with identical observable outputs. For the security theorem, we formally reduce the security of our scheme against semi-honest (or "honest but curious") adversaries to the semi-honest security of the 2PC backend. The theorem provides protection against side-channels arising from conditionals and memory access patterns. We also prove a formal security theorem against semi-honest adversaries for SCP (Section IV and Section V).

5) Evaluation: We have implemented EzPC using ABY [18] as the cryptographic backend. We compare EzPC with prior compilers in Section I-D and with specialized protocols in Section VII. We evaluate EzPC by implementing a wide range of secure prediction benchmarks including linear and naïve Bayes classifiers, decision trees, deep neural networks, state-of-the-art classifiers from Tensorflow [27] and BONSAI [28], and also matrix factorization from Nikolaenko et al. [25]. Our results demonstrate three key points. First, EzPC makes it convenient for general programmers to write 2PC protocols. The lines of EzPC code are comparable to the lines of C++ or Java code required to describe the functionality to be computed. Second, the performance of the protocols generated by EzPC is comparable to or better than (up to 19x)

[2]Using swap and disk space is feasible but it causes huge slowdown (Figure 15).

their state-of-the-art, hand-crafted implementations. Finally, we demonstrate the usefulness of SCP by implementing an application that requires more than 300 million gates (Section VI and Section VII).

Summary. EzPC is carefully designed to enable non-cryptographers (and cryptographers alike) to write efficient 2PC protocols for ML algorithms. With the novel combination of language design, cryptographic cost aware compiler, and scalability through SCP, programmers can program in a high-level language absent of all the cryptographic details, while the compiler chooses the best circuit representation (boolean or arithmetic) for different parts of the program and also handles interconversions between them – all this in a provably secure manner. Without EzPC, implementing 2PC protocols for the wide array of ML algorithms from Section VII would have been a Herculean task and has never been done.

B. Formalization

The need for formalization in, and proving correctness of, a compiler for secure computation, cannot be overemphasized. Systems such as CBMC-GC [9] and HyCC [29], claim to translate ANSI-C to cryptographic protocols without a formal model, which can be quite problematic to the user. It is unknown whether the system would support a specific feature of ANSI-C or not. As a result, the user of such a system is left to guess which language features she can or cannot use. For example, none of these works mention how they handle function pointers, undefined behaviors, and many other vagaries of ANSI-C, but remark that they handle "some" pointer arithmetic [29]. Hence, a user who writes code in these systems using pointers, would have to speculate whether her code is supported or not: the declaration `int x(int[2])` is probably supported but `char (*(*x[3])())[5]` is probably not, even though both are natural in ANSI-C. In addition, it is unknown what would happen if a user were to erroneously write a program that dereferences arrays beyond their bounds.

On the other hand, EzPC provides a clean programming interface and everything the user can express in the language is completely supported. When the user writes an incorrect program, the type checker guides her to a correct implementation. In particular, the compiler catches all the buffer overflow errors and reports them at compile time. The formal semantics completely specify how each program is evaluated and there are no surprises. Although proving correctness of the compilation process has distinct advantages, formalization is non-trivial when multiple protocols interact with each other. This work provides the first such formal model, majorly differentiating itself from informally specified systems like HyCC [29] that leave users to guesswork.

C. Related Work

Compilers such as Wysteria [7], ObliVM [8], and CBMC-GC [9] use backends with single representations (either boolean or arithmetic), and so the challenges associated with multiple representations and interconversions are absent.

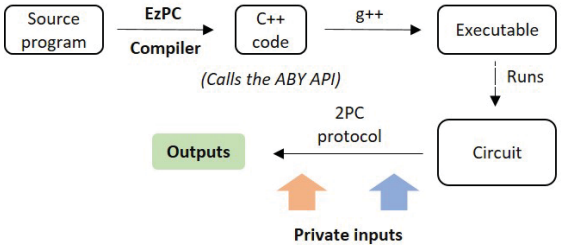

Fig. 2: EzPC toolchain

Moreover, as shown in Section I-D, they perform poorly on ML tasks that involve a mix of arithmetic and boolean computations. Before ABY [18], several works have proposed combining secure computation protocols based on homomorphic encryption and Yao's garbled circuits (e.g. [3], [30], [31], [32], [33], [25], [34], [35]), and some have also developed tools that allow writing such combinations (e.g. [22], [23], [19], [20], [24]). However, as Demmler et al. [18] observe, due to the high conversion cost between homomorphic encryption and Yao's garbled circuits, these combined protocols do not gain much performance over a single protocol. Additionally, these prior works are low-level and provide informal languages or libraries that lack formal semantics and static guarantees. We provide a detailed survey of related work in Section VIII.

D. Empirical comparison with prior frameworks

We show that it is critical to use a mix of arithmetic and boolean circuits for performance. Previous works have observed that Yao's garbled circuits do not scale to machine learning examples that require a large number of multiplications [18], [19], [20], [5], [4], [21]. Indeed, this is one of the main drivers behind the development of various specialized 2PC protocols in previous works [5], [4], [1] (we compare against them in Section VII) . Here, we empirically demonstrate the performance benefits of mixed computations over boolean-only compute by comparing with the state-of-the-art Yao-based compilers CBMC-GC [9] and ObliVM [8].

The largest benchmark in CBMC-GC is a multiplication of two 8×8 matrices for which it generates about a million gates and takes about 10 seconds to run. In contrast, EzPC uses arithmetic sharing, generates 1218 gates, and runs in less than 0.1 seconds. On the same benchmark, EzPC is over two orders of magnitude faster than ARM2GC [36]. When we tried multiplying two 20×20 matrices with CBMC-GC, it timed out after 5 hours. Our benchmarks require much bigger computations (e.g., multiplying a 64×576 matrix with a 576×1024 matrix). Unlike CBMC-GC, ObliVM can scale to larger benchmarks through Yao-pipelining [37]. We evaluated the program in Figure 1 with vectors of lengths varying between 10^3 and 10^6 using both EzPC and ObliVM. EzPC evaluates the inner-product in arithmetic and, empirically, is at least 25x faster than ObliVM on the same hardware.

```
1  //circuit builders for arithmetic and boolean
   Circuit *ycirc = s[S_YAO] -> GetCircuitBuildRoutine();
3  Circuit *acirc = s[S_ARITH] -> GetCircuitBuildRoutine();
   ...
5  if(role == SERVER) {
     //Put gates to read w and b
7  } else { //role == CLIENT
     //Put gates to read x
9  }

11 for(uint32_t i = 0; i < 30; i = i + 1) { //acc = w^T x
     share *a_t_0 = acirc -> PutMULGate(a_w[i], a_x[i]);
13   a_acc = acirc -> PutADDGate(a_acc, a_t_0);
   }
15
   //convert acc and b from arithmetic to boolean
17 share *y_acc = ycirc -> PutA2YGate(a_acc);

19 share *y_pred = ycirc -> PutGTGate(y_acc, y_b);
   uint32_t one = 1;
21 share *y_1 = ycirc -> PutCONSGate(one, bitlen);
   uint32_t zero = 0;
23 share *y_0 = ycirc -> PutCONSGate(zero, bitlen);
   share *y_t = ycirc -> PutMUXGate(y_pred, y_1, y_0);
25
   share *y_out = ycirc -> PutOUTGate(y_t, CLIENT);
27 party -> ExecCircuit();

29 if(role == CLIENT){ //only to the client
     uint32_t_o = y_out -> get_clear_value<uint32_t>();
31 }
```

Fig. 3: EzPC compiler (partial) output for Figure 1

II. EzPC Overview

Figure 2 shows an overview of the EzPC toolchain. We give a brief overview of each of these phases below.

Source language. Consider the example $w^T x > b$ from Section I, where w and b are the server's input (a classifier) and x is the client's input vector. Figure 1 shows the EzPC code for this example. The code first reads the server's (resp. client's) input using `input1` (resp. `input2`). It then uses a `for` loop to compute `acc`, the inner product of `w` and `x`. Finally, the code compares `acc` with `b` and outputs this only to the client using `output2`.

EzPC source language is a simple, imperative language that enables the programmers to express 2PC computations in terms of their "ideal" functionalities, without dealing with any cryptographic details. The language provides multi-dimensional arrays with public indices, conditional expressions (the ternary ? : operator), `for` loops, `if` statements, and syntax for input/output from each party.

EzPC compiler. EzPC compiler takes as input a source program and produces a C++ program as output. Figure 3 shows (partial) output code for the example in Figure 1 – this is also how a program written directly in ABY would look like. The output program contains party-specific code for inputs and outputs (`role == SERVER` and `role == CLIENT`), and common code for the computation.

The compiler splits the input program into *public* and *secret* components. The public components translate into regular C++ code, while the secret components translate into API calls into our 2PC backend (ABY). For example, in Figure 1, the EzPC compiler realizes that the array index `i` in the inner product loop is public, and hence the access locations need not be hidden. Therefore, it compiles the `for` loop into a C++ `for` loop that will be executed in-clear (line 11).

Within the secret components, the EzPC compiler is "cryptographic cost aware", and appropriately picks either arithmetic or boolean circuit representations for different sub-components. For example, the compiler realizes that the inner product computation is more efficient in the arithmetic representation, and therefore it builds the corresponding circuit using the arithmetic circuit builder `acirc` (lines 12 and 13). On the other hand, since the comparison with `b`, and the conditional expression (the '? :' operator) computation are more efficient in the boolean representation, the EzPC compiler uses the Yao circuit builder `ycirc` to build the corresponding comparison and multiplexer circuits (lines 19 to 24).

Using both arithmetic and boolean representations requires conversions between them. The EzPC compiler also instruments these conversion gates accordingly. For example, on line 17, the compiler converts `a_acc` to a boolean representation, before it is input to the comparison and multiplexer circuits.

Circuit generation and evaluation. The next step is to compile the output C++ code and execute it. Doing so evaluates away the public parts of the program, including the array accesses, and generates a circuit comprising of arithmetic and boolean gates, with appropriate conversion gates. The circuit is then evaluated using a 2PC protocol.

Advantages of EzPC. We can now concretely see the advantages of EzPC. Unarguably, it is easier for a developer to program and get the code right in Figure 1, rather than the code in Figure 3. EzPC also enables the programmer to easily modify their code, while the compiler takes care of efficiency. For example, consider in Figure 1 a change from multiplication to bitwise-or in the `for` loop. It turns out that in this case, it is more efficient to do both the addition and bitwise-or using boolean circuits (if the addition is done using arithmetic, the conversion cost starts to take over). In EzPC, the programmer simply needs to change one operator in the source code, and the compiler generates efficient code that uses boolean addition. Whereas, if the programmer was writing ABY code, she either has to sacrifice performance, or would have to revisit many parts of the circuit and change them. In summary, EzPC raises the level of abstraction for the programmer, and generates efficient 2PC protocols automatically, while its metatheory provides strong correctness and security guarantees.

Comparison with ABY. We have evaluated against handwritten ABY code for the examples in the ABY repository and EzPC has comparable performance. However, these examples are not representative of practical ML tasks. Writing sophisticated ML applications such as neural networks in ABY is a very tedious and nearly-impossible engineering task even for crypto experts – automating the same is our main contribution.

III. Cryptographic cost awareness

In this section, we explain various heuristics that EzPC uses to split the computation in a program into arithmetic and boolean parts. Since finding an optimum split is an NP-hard problem (the predicates in `if` statements can have arbitrary non-linear arithmetic), EzPC uses heuristics that perform well in practice (Section VII).

The split between arithmetic and boolean requires knowing the cost of individual operations (addition, multiplication, interconversion between arithmetic and boolean, etc.). Demmler et al. [18] document these costs by running microbenchmarks for basic operations and interconversions (Figures 2 and 3 in [18]). EzPC heuristics are based on their results.

Converting an arithmetic share to a boolean share requires computing a garbled circuit for addition. The size of this circuit grows linearly in the bit-width of the inputs. Similarly, converting from boolean to arithmetic requires computing a garbled circuit for subtraction, which is also linear. Since each conversion has roughly the same cost as a boolean addition, EzPC performs addition using a boolean circuit if the operands are boolean shared, else it uses an arithmetic circuit.

For multiplication, EzPC always chooses an arithmetic circuit, as the cost of a boolean multiplication is much higher than the cost of converting the operands from boolean to arithmetic, performing an arithmetic multiplication, and then converting the result back to boolean ($\sim$ 9x more time and $\sim$ 20x more communication in a LAN setting when using garbled circuits[3]). The size of a boolean multiplication circuit is quadratic, which causes this performance gap. Roughly, multiplying two 64-bit integers using arithmetic sharing requires only 2 multiplications, whereas Yao requires at least 4096 AES operations in the online phase. Since this gap is quite large, we believe that this choice is optimal for realistic network settings. Finally, EzPC chooses boolean circuits for all the operations lacking arithmetic support in ABY, e.g. comparisons, bit-shifts, etc. We provide more implementation details in Section VI.

IV. Formal development

In this section we describe the syntax of EzPC and prove correctness and security of the compiler. We first formalize our source language (an example program being Figure 1), and its runtime semantics. This semantics describes the "trusted third party" execution semantics of the source programs and generates observations corresponding to the values revealed to the parties. We then present the compilation rules that type check a program in the source language and generate a program in the intermediate language (an example program being Figure 3). Next, we present the runtime semantics of our intermediate language that evaluates to a circuit by "evaluating away" the public parts and the arrays. Crucially, this step does not have access to the secret inputs; those are processed by our distributed circuit semantics that models the 2PC backend. Evaluation in this distributed setting involves the parties running an interactive protocol. This step, like the source semantics, emits observations corresponding to the values revealed to the parties.

To prove the correctness of EzPC, we prove that the observations in source semantics and the distributed circuit semantics are identical (Theorem 1). We combine this correctness theorem and the security of the 2PC backend to prove security of the protocols generated by EzPC (Theorem 2). We present the representative parts of our formalization.

$$
\begin{array}{rrcl}
\text{Base type} & \sigma & ::= & \texttt{uint} \mid \texttt{bool} \\
\text{Type} & \psi & ::= & \sigma \mid \sigma[n] \\
\text{Constant} & c & ::= & n \mid \top \mid \bot \\
\text{Expression} & e & ::= & c \mid x \mid e_1 \times e_2 \mid e_1 > e_2 \mid e_1 \,?\, e_2 \,:\, e_3 \\
 & & \mid & [\overline{e_i}]_n \mid x[e] \mid \texttt{in}_j \\
\text{Statement} & s & ::= & \psi\, x = e \mid x := e \mid \texttt{for}\ x\ \texttt{in}\ [n_1, n_2]\ \texttt{do}\ s \\
 & & \mid & x[e_1] := e_2 \mid \texttt{if}(e, s_1, s_2) \mid \texttt{out}\ e \mid s_1; s_2 \\
 & & \mid & \texttt{while}\ x \leq n\ \texttt{do}\ s
\end{array}
$$

Fig. 4: Source language syntax

Source language. Our language is a simple imperative language shown in Figure 4. Types ψ consist of the base types σ, and arrays of base types $\sigma[n]$, where n is the array length. Expressions e in the language include the integer constants n, `bool` constants $\top$ and $\bot$, variables x, binary operations $e_1 \times e_2$ and $e_1 > e_2$, conditionals $e\,?\,e_1 : e_2$, array literals $[\overline{e_i}]_n$[4], and array reads $x[e]$. The expression $\texttt{in}_j$ denotes input from party j. The statements s in the language comprise of variable declarations, assignments, `for` loops, array writes, `if` statements, and sequence of statements. The statement `out` e denotes revealing the value of e to the parties. The `while` statement is an internal syntax that is not exposed to the programmer. The full EzPC language has additional features that are discussed in Section IV-A. The readers who are interested in using the compiler as a black box can move directly to Section IV-A without loss of continuity.

Source semantics. The runtime semantics for the source language is shown in Figure 5. These semantics show how a "trusted third party" computes the outputs when given the inputs of both the parties. Values v, runtime environments ρ, and observations O are defined as follows:

$$
\begin{array}{rrcl}
\text{Value} & v & ::= & c \mid [\overline{c_i}]_n \\
\text{Runtime environment} & \rho & ::= & \cdot \mid \rho, x \mapsto v \\
\text{Observation} & O & ::= & \cdot \mid c, O
\end{array}
$$

Values consist of constants and array of constants, runtime environment ρ maps variables to values, and observations are sequences of constants.

The judgment $\rho \vdash e \Downarrow v$ denotes the big-step evaluation of an expression e to a value v under the runtime environment ρ. Rule (E-VAR) looks up the value of x in the environment. Rule (E-MULT) inductively evaluates e_1 and e_2, and returns their product. Rule (E-READ) evaluates an array read operation. It first evaluates x to an array value $[\overline{c_i}]_{n_1}$, and e to a `uint` value n. It then returns c_n, the n-th index value in the array, provided

[3]Using boolean computation based on [14] was slower than garbled circuits.

[4]We write $\overline{e}$ (and similarly for other symbols) to denote a sequence of expressions. The length of the sequence is usually clear from the context.

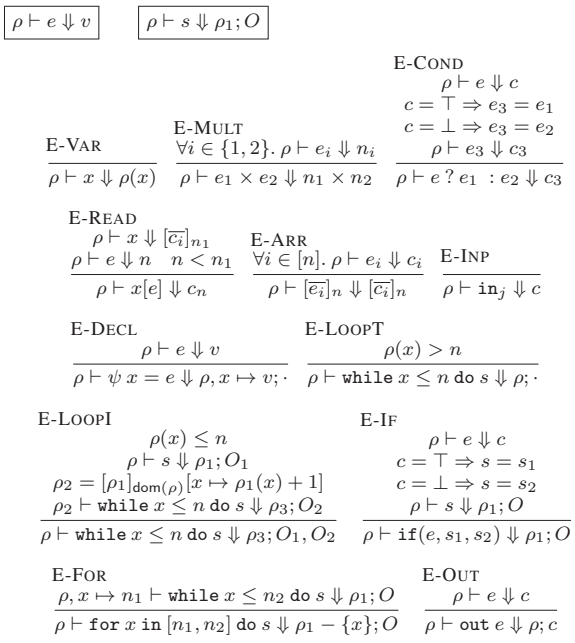

Fig. 5: Source semantics

$n < n_1$, the length of the array. Rule (E-INP) evaluates to some constant c denoting party j's input. An array input can be written in the language as $[\mathtt{in}_j]_n$, which can then evaluate using the rule (E-ARR) (the notation $\forall i \in [n]$ is read as $\forall i \in \{0 \ldots n-1\}$). The remaining rules are straightforward, and are elided for space reasons.

The judgment $\rho \vdash s \Downarrow \rho_1; O$ represents the big-step evaluation of a statement s under environment ρ, producing a new environment ρ_1 and observations O. Rule (E-DECL) evaluates the expression e to v, and returns the updated environment $\rho, x \mapsto v$, with empty observations. The `for` statements evaluate through the internal `while` syntax. Specifically, the rule (E-FOR) appends ρ with $x \mapsto n_1$, evaluates $\mathtt{while}\, x \leq n_2\, \mathtt{do}\, s$ to $\rho_1; O$, and returns $\rho_1 - \{x\}$ (removing x from ρ_1) and O. Rule (E-LOOPI) shows the inductive case for `while` statements, when $\rho(x) \leq n$. The rule evaluates s, producing $\rho_1; O_1$. It then restricts ρ_1 to the domain of ρ ($[\rho_1]_{\mathsf{dom}(\rho)}$) to remove the variables added by s, increments the value of x, and evaluates the `while` statement under this updated environment. Rule (E-LOOPT) is the termination case for `while`, when $\rho(x) > n$. Finally, the rule (E-OUT) evaluates the expression, and adds its value to the observations.

Intermediate language. Figure 6 shows the intermediate language of our compiler. The syntax follows that of the source language, except that the types and operators are *labeled*. A label ℓ can be the public label $\mathcal{P}$ or one of the secret labels $\mathcal{A}$ or $\mathcal{B}$, which denote arithmetic and boolean respectively. Types τ are then labeled base types σ^ℓ and arrays of labeled base types $\sigma^\ell[n]$. Most of the expression forms $\widetilde{e}$ are same as e, except that the binary operators, and

Secret label	m	::=	$\mathcal{A} \mid \mathcal{B}$
Label	ℓ	::=	$\mathcal{P} \mid m$
Type	τ	::=	$\sigma^\ell \mid \sigma^\ell[n]$
Expression	$\widetilde{e}$	::=	$c \mid x \mid \widetilde{e}_1 \times_\ell \widetilde{e}_2 \mid \widetilde{e}_1 >_\ell \widetilde{e}_2 \mid x[\widetilde{e}] \mid [\overline{\widetilde{e}_i}]_n$
		$\mid$	$\widetilde{e}\, ?_\ell\, \widetilde{e}_1\ : \widetilde{e}_2 \mid \mathtt{in}_j^m \mid \langle \ell \rhd m \rangle\, \widetilde{e}$
Statement	$\widetilde{s}$	::=	$\tau\ x = \widetilde{e} \mid x := \widetilde{e} \mid \cdots \mid \widetilde{s}_1; \widetilde{s}_2 \mid \ldots$

Fig. 6: Intermediate language syntax

the conditional forms are annotated with labels ℓ. Looking ahead, the label determines how the operators are evaluated: $\mathcal{P}$-labeled operators are evaluated in-clear, $\mathcal{A}$-labeled operators generate arithmetic circuits, and $\mathcal{B}$-labeled operators generate boolean circuits. The form $\langle \ell \rhd m \rangle\, \widetilde{e}$ denotes coercing $\widetilde{e}$ from label ℓ to label m.

Source to intermediate compilation. We provide the compilation rules in Figure 7. We present the rules in a declarative style, where the rules are non-syntax directed, and the labels ℓ are chosen non-deterministically. Section VI describes the label inference scheme in our implementation.

The judgment $\Gamma \vdash e : \tau \rightsquigarrow \widetilde{e}$, where Γ maps variables x to types τ, says that under Γ, e (in the source language) compiles to $\widetilde{e}$ (in the intermediate language), where $\widetilde{e}$ has type τ. Rules (T-UINT) and (T-BOOL) assigns the label $\mathcal{P}$ to the constants, as the constants are always public. Rule (T-MULT) compiles a multiplication to either a public multiplication ($\times_\mathcal{P}$), or a secret arithmetic multiplication ($\times_\mathcal{A}$). As our compiler is cryptographic cost aware, it never compiles the multiplication to boolean multiplication $\times_\mathcal{B}$ (Section VI). In a similar manner, rule (T-GT) compiles $e_1 > e_2$ to either public comparison, or secret boolean comparison $>_\mathcal{B}$ (and never $>_\mathcal{A}$). The rule for conditional (T-COND) has two cases: when the conditional expression e is of type $\mathtt{bool}^\mathcal{P}$, both the branches have a base type σ^{ℓ_1}, for an arbitrary ℓ_1, and the conditional is compiled to a public conditional, whereas when the conditional expression has type $\mathtt{bool}^\mathcal{B}$, ℓ_1 is also $\mathcal{B}$, and the conditional is compiled to a secret conditional using a boolean circuit. Note that we restrict the type of the branches to be of base type. Rule (T-READ) type checks an array read. It checks that the array index e is public, and uses a static bounds checking judgment $\Gamma \models e < n$ to prove that the array index is in bounds [5]. Secret dependent array accesses can be encoded using the conditional expressions (see Section VI). Rule (T-INP) picks a label m for the input. Finally, the rule (T-SUB) is the subsumption rule that coerces an expression of type σ^ℓ to an expression of type σ^m using the coerce expression. It is important for security that the secrets cannot be coerced to public values and indeed (T-SUB) does not permit it.

Judgment $\Gamma \vdash s : \tau \rightsquigarrow \widetilde{s} \mid \Gamma_1$ compiles a statement s resulting in the statement $\widetilde{s}$ and type environment Γ_1. Rule (T-DECL) picks a label ℓ, and adds the binding for x to the environment (if $\psi = \sigma$, $\psi^\ell = \sigma^\ell$, else if $\psi = \sigma[n]$, $\psi^\ell = \sigma^\ell[n]$). Rule (T-ASSIGN) looks up the type of x in Γ and compiles e to $\tilde{e}$ of same type. Note that in this rule we

[5] Section VI discusses our implementation of this check.

$\boxed{\Gamma \vdash e : \tau \rightsquigarrow \widetilde{e}}$ $\boxed{\Gamma \vdash s \rightsquigarrow \widetilde{s} \mid \Gamma_1}$

$$\textsc{T-UInt}\ \frac{}{\Gamma \vdash n : \mathtt{uint}^{\mathcal{P}} \rightsquigarrow n} \qquad \textsc{T-Bool}\ \frac{c = \top \vee c = \bot}{\Gamma \vdash c : \mathtt{bool}^{\mathcal{P}} \rightsquigarrow c} \qquad \textsc{T-Inp}\ \frac{}{\Gamma \vdash \mathtt{in}_j : \sigma^m \rightsquigarrow \mathtt{in}_j^m}$$

$$\textsc{T-Mult}\ \frac{\forall i \in \{1,2\}.\, \Gamma \vdash e_i : \mathtt{uint}^{\ell} \rightsquigarrow \widetilde{e}_i \quad (\ell = \mathcal{P}) \vee (\ell = \mathcal{A})}{\Gamma \vdash e_1 \times e_2 : \mathtt{uint}^{\ell} \rightsquigarrow \widetilde{e}_1 \times_{\ell} \widetilde{e}_2} \qquad \textsc{T-Gt}\ \frac{\forall i \in \{1,2\}.\, \Gamma \vdash e_i : \mathtt{uint}^{\ell} \rightsquigarrow \widetilde{e}_i \quad (\ell = \mathcal{P}) \vee (\ell = \mathcal{B})}{\Gamma \vdash e_1 > e_2 : \mathtt{bool}^{\ell} \rightsquigarrow \widetilde{e}_1 >_{\ell} \widetilde{e}_2}$$

$$\textsc{T-Read}\ \frac{\Gamma \vdash x : \sigma^{\ell}[n] \rightsquigarrow x \quad \Gamma \vdash e : \mathtt{uint}^{\mathcal{P}} \rightsquigarrow \widetilde{e} \quad \Gamma \models e < n}{\Gamma \vdash x[e] : \sigma^{\ell} \rightsquigarrow x[\widetilde{e}]} \qquad \textsc{T-Cond}\ \frac{\Gamma \vdash e : \mathtt{bool}^{\ell} \rightsquigarrow \widetilde{e} \quad \forall i \in \{1,2\}.\, \Gamma \vdash e_i : \sigma^{\ell_1} \rightsquigarrow \widetilde{e}_i \quad \ell = \mathcal{P} \vee (\ell = \mathcal{B} \wedge \ell_1 = \mathcal{B})}{\Gamma \vdash e\,?\,e_1\ :e_2 : \sigma^{\ell_1} \rightsquigarrow \widetilde{e}\,?_{\ell}\,\widetilde{e}_1\ :\widetilde{e}_2}$$

$$\textsc{T-Arr}\ \frac{\forall i \in [n].\, \Gamma \vdash e_i : \sigma^{\ell} \rightsquigarrow \widetilde{e}_i}{\Gamma \vdash [\overline{e_i}]_n : \sigma^{\ell}[n] \rightsquigarrow [\overline{\widetilde{e}_i}]_n} \qquad \textsc{T-Sub}\ \frac{\Gamma \vdash e : \sigma^{\ell} \rightsquigarrow \widetilde{e}}{\Gamma \vdash e : \sigma^{m} \rightsquigarrow \langle \ell \rhd m \rangle\, \widetilde{e}}$$

$$\textsc{T-Decl}\ \frac{\Gamma \vdash e : \psi^{\ell} \rightsquigarrow \widetilde{e}}{\Gamma \vdash \psi\ x = e \rightsquigarrow \psi^{\ell}\ x = \widetilde{e} \mid \Gamma, x : \psi^{\ell}} \qquad \textsc{T-Assgn}\ \frac{\Gamma(x) = \sigma^{\ell} \quad \Gamma \vdash e : \sigma^{\ell} \rightsquigarrow \widetilde{e}}{\Gamma \vdash x := e \rightsquigarrow x := \widetilde{e} \mid \Gamma}$$

$$\textsc{T-For}\ \frac{\Gamma, x : \mathtt{uint}^{\mathcal{P}} \vdash \mathtt{while}\ x \leq n_2\ \mathtt{do}\ s \rightsquigarrow \mathtt{while}\ x \leq n_2\ \mathtt{do}\ \widetilde{s} \mid \_}{\Gamma \vdash \mathtt{for}\ x\ \mathtt{in}\ [n_1, n_2]\ \mathtt{do}\ s \rightsquigarrow \mathtt{for}\ x\ \mathtt{in}\ [n_1, n_2]\ \mathtt{do}\ \widetilde{s} \mid \Gamma}$$

$$\textsc{T-Write}\ \frac{\Gamma \vdash x : \sigma^{\ell}[n] \rightsquigarrow x \quad \Gamma \vdash e_1 : \mathtt{uint}^{\mathcal{P}} \rightsquigarrow \widetilde{e_1} \quad \Gamma \vdash e_2 : \sigma^{\ell} \rightsquigarrow \widetilde{e_2} \quad \Gamma \models e_1 < n}{\Gamma \vdash x[e_1] := e_2 \rightsquigarrow x[\widetilde{e_1}] := \widetilde{e_2} \mid \Gamma} \qquad \textsc{T-Out}\ \frac{\Gamma \vdash e : \sigma^{m} \rightsquigarrow \widetilde{e}}{\Gamma \vdash \mathtt{out}\ e \rightsquigarrow \mathtt{out}\ \widetilde{e} \mid \Gamma}$$

$$\textsc{T-If}\ \frac{\Gamma \vdash e : \mathtt{bool}^{\mathcal{P}} \rightsquigarrow \widetilde{e} \quad \forall i \in \{1,2\}.\, \Gamma \vdash s_i \rightsquigarrow \widetilde{s_i} \mid \_}{\Gamma \vdash \mathtt{if}(e, s_1, s_2) \rightsquigarrow \mathtt{if}(\widetilde{e}, \widetilde{s_1}, \widetilde{s_2}) \mid \Gamma} \qquad \textsc{T-Seq}\ \frac{\Gamma \vdash s_1 \rightsquigarrow \widetilde{s}_1 \mid \Gamma_1 \quad \Gamma_1 \vdash s_2 \rightsquigarrow \widetilde{s}_2 \mid \Gamma_2}{\Gamma \vdash s_1; s_2 \rightsquigarrow \widetilde{s}_1; \widetilde{s}_2 \mid \Gamma_2}$$

$$\textsc{T-While}\ \frac{\Gamma(x) = \mathtt{uint}^{\mathcal{P}} \quad \Gamma \vdash s \rightsquigarrow \widetilde{s} \mid \_ \quad x \notin \mathsf{modifies}(s)}{\Gamma \vdash \mathtt{while}\ x \leq n_2\ \mathtt{do}\ s \rightsquigarrow \mathtt{while}\ x \leq n_2\ \mathtt{do}\ \widetilde{s} \mid \Gamma}$$

Fig. 7: Source compilation

restrict the type of variable x to be of base type. Rule (T-For) adds the loop counter x to Γ at type $\mathtt{uint}^{\mathcal{P}}$, and delegates type checking to the `while` form. Rule (T-Out) types the expression e at some secret label m. Rule (T-If) checks that the conditional expression is public (for secret conditional expressions see Section IV-A), and rule (T-Seq) sequences the type environments. Finally, the typing rule for the (internal) `while` form ensures that x is mapped in Γ at type $\mathtt{uint}^{\mathcal{P}}$, and that the statement s does not modify x ($x \notin \mathsf{modifies}(s)$) – this is necessary for ensuring termination.

As mentioned earlier, the intermediate language models the code, such as in Figure 3, output by our compiler. Next, a program in the intermediate language is evaluated to a circuit that can be executed in the distributed runtime later. The evaluation to a circuit computes away the public parts of the

$$\begin{array}{rrcl}
\text{Wire id} & w & & \\
\text{Circuit gate} & g & ::= & w \mid \mathtt{in}_j^m \mid \mathtt{mult}\ g_1\ g_2 \mid \mathtt{gt}\ g_1\ g_2 \\
 & & \mid & \mathtt{mux}\ g\ g_1\ g_2 \mid \langle \ell \rhd m \rangle\ g \mid c \\
\text{Sub-circuit} & \widetilde{v} & ::= & g \mid [\overline{g_i}]_n \\
\text{Circuit} & \chi & ::= & \cdot \mid \mathtt{bind}\ g\ w \mid \mathtt{out}\ g \mid \chi_1; \chi_2
\end{array}$$

Fig. 8: Circuits syntax

program and also *flattens* the arrays so that the circuits are unaware of the array structure. Crucially, this phase of the semantics does not have access to the secret inputs. Below, we first provide the language for the circuits followed by the evaluation rules.

Evaluation to Circuits. Figure 8 shows the syntax of circuits. A wire id range w denotes a set of circuit wires that carry the runtime value of a variable with a secret label (we will concretely define these runtime values later as part of the circuit semantics). Circuit gates g are wires w, input gates $\mathtt{in}_j^m$, multiplication gates `mult`, comparison gates `gt`, and multiplexer `mux` gates, coerce gates $\langle \ell \rhd m \rangle$, and constants. Sub-circuits $\widetilde{v}$ (generated from $\widetilde{e}$) then consist of gates and arrays of gates. A circuit χ is either empty, `bind`-ing of a circuit gate g to wire w, `out` gate, or a sequence of circuits.

$\boxed{\widetilde{\rho} \vdash \widetilde{e} \Downarrow \widetilde{v}}$ $\boxed{\widetilde{\rho} \vdash \widetilde{s} \Downarrow \widetilde{\rho}_1; \chi}$

$$\textsc{S-Var}\ \frac{}{\widetilde{\rho} \vdash x \Downarrow \widetilde{\rho}(x)} \qquad \textsc{S-PMult}\ \frac{\forall i \in \{1,2\}.\, \widetilde{\rho} \vdash \widetilde{e}_i \Downarrow n_i}{\widetilde{\rho} \vdash \widetilde{e}_1 \times_{\mathcal{P}} \widetilde{e}_2 \Downarrow n_1 \times n_2} \qquad \textsc{S-Read}\ \frac{\widetilde{\rho} \vdash x \Downarrow [\overline{w_i}]_{n_1} \quad \widetilde{\rho} \vdash \widetilde{e} \Downarrow n \quad n < n_1}{\widetilde{\rho} \vdash x[\widetilde{e}] \Downarrow w_n}$$

$$\textsc{S-SMult}\ \frac{\forall i \in \{1,2\}.\, \widetilde{\rho} \vdash \widetilde{e}_i \Downarrow g_i}{\widetilde{\rho} \vdash \widetilde{e}_1 \times_{\mathcal{A}} \widetilde{e}_2 \Downarrow \mathtt{mult}\ g_1\ g_2} \qquad \textsc{S-SGt}\ \frac{\forall i \in \{1,2\}.\, \widetilde{\rho} \vdash \widetilde{e_i} \Downarrow g_i}{\widetilde{\rho} \vdash \widetilde{e_1} >_{\mathcal{B}} \widetilde{e_2} \Downarrow \mathtt{gt}\ g_1\ g_2}$$

$$\textsc{S-SCond}\ \frac{\forall i \in \{1,2,3\}.\, \widetilde{\rho} \vdash \widetilde{e}_i \Downarrow g_i}{\widetilde{\rho} \vdash \widetilde{e}_1\ ?_{\mathcal{B}}\ \widetilde{e}_2\ :\widetilde{e}_3 \Downarrow \mathtt{mux}\ g_1\ g_2\ g_3} \qquad \textsc{S-Coerce}\ \frac{\widetilde{\rho} \vdash \widetilde{e} \Downarrow g}{\widetilde{\rho} \vdash \langle \ell \rhd m \rangle\, \widetilde{e} \Downarrow \langle \ell \rhd m \rangle\, g}$$

$$\textsc{S-PCond}\ \frac{\widetilde{\rho} \vdash \widetilde{e} \Downarrow c \quad c = \top \Rightarrow \widetilde{e}_3 = \widetilde{e}_1 \quad c = \bot \Rightarrow \widetilde{e}_3 = \widetilde{e}_2 \quad \widetilde{\rho} \vdash \widetilde{e}_3 \Downarrow \widetilde{v}}{\widetilde{\rho} \vdash \widetilde{e}\,?_{\mathcal{P}}\,\widetilde{e}_1\ :\widetilde{e}_2 \Downarrow \widetilde{v}} \qquad \textsc{S-Inp}\ \frac{}{\widetilde{\rho} \vdash \mathtt{in}_j^m \Downarrow \mathtt{in}_j^m}$$

$$\textsc{S-DeclC}\ \frac{\widetilde{\rho} \vdash \widetilde{e} \Downarrow \widetilde{v} \quad (\widetilde{v} = c) \vee (\widetilde{v} = [\overline{c_i}]_n)}{\widetilde{\rho} \vdash \tau\ x = \widetilde{e} \Downarrow \widetilde{\rho}, x \mapsto \widetilde{v}; \cdot} \qquad \textsc{S-DeclCkt}\ \frac{\widetilde{\rho} \vdash \widetilde{e} \Downarrow g \quad \mathsf{fresh}\ w}{\widetilde{\rho} \vdash \tau\ x = \widetilde{e} \Downarrow \widetilde{\rho}, x \mapsto w; \mathtt{bind}\ g\ w}$$

$$\textsc{S-DeclCktA}\ \frac{\widetilde{\rho} \vdash \widetilde{e} \Downarrow [\overline{g_i}]_n \quad \forall i \in [n].\, \mathsf{fresh}\ w_i}{\widetilde{\rho} \vdash \tau\ x = \widetilde{e} \Downarrow \widetilde{\rho}, x \mapsto [\overline{w_i}]_n; \overline{\mathtt{bind}\ g_i\ w_i}} \qquad \textsc{S-Out}\ \frac{\widetilde{\rho} \vdash \widetilde{e} \Downarrow g}{\widetilde{\rho} \vdash \mathtt{out}\ \widetilde{e} \Downarrow \widetilde{\rho}; \mathtt{out}\ g}$$

$$\textsc{S-If}\ \frac{\widetilde{\rho} \vdash \widetilde{e} \Downarrow c \quad c = \top \Rightarrow \widetilde{s} = \widetilde{s_1} \quad c = \bot \Rightarrow \widetilde{s} = \widetilde{s_2} \quad \widetilde{\rho} \vdash \widetilde{s} \Downarrow \widetilde{\rho}_1; \chi}{\widetilde{\rho} \vdash \mathtt{if}(\widetilde{e}, \widetilde{s_1}, \widetilde{s_2}) \Downarrow \widetilde{\rho}_1; \chi} \qquad \textsc{S-WriteCkt}\ \frac{\widetilde{\rho} \vdash x \Downarrow [\overline{w_i}]_n \quad \widetilde{\rho} \vdash \widetilde{e}_1 \Downarrow n_1 \quad n_1 < n \quad \mathsf{fresh}\ w \quad \widetilde{\rho} \vdash \widetilde{e}_2 \Downarrow g \quad \widetilde{\rho}_1 = \widetilde{\rho}[x \mapsto ([\overline{w_i}]_n[n_1 \mapsto w])]}{\widetilde{\rho} \vdash x[\widetilde{e}_1] := \widetilde{e}_2 \Downarrow \widetilde{\rho}_1; \mathtt{bind}\ g\ w}$$

Fig. 9: Evaluation of Intermediate Language to Circuit

Figure 9 shows the judgments for the evaluation of the intermediate language to a circuit. The circuit generation environment maps variables to sub-circuits:

$$\text{Circuit generation environment} \quad \widetilde{\rho} \quad ::= \quad \cdot \mid \widetilde{\rho}, x \mapsto \widetilde{v}$$

We first focus on the expression evaluation judgment $\widetilde{\rho} \vdash \widetilde{e} \Downarrow \widetilde{v}$. Rules (S-PMULT) and (S-SMULT) illustrate the significance of the operator labels. In particular, the rule (S-PMULT) evaluates a public multiplication $\widetilde{e}_1 \times_{\mathcal{P}} \widetilde{e}_2$ to $n_1 \times n_2$, similar to the source semantics of Figure 5. In contrast, the rule (S-SMULT) evaluates a secret multiplication $\widetilde{e}_1 \times_{\mathcal{A}} \widetilde{e}_2$ to an arithmetic multiplication gate $\texttt{mult}\ g_1\ g_2$. As mentioned above, the intermediate language expressions generated by our compiler never have $\widetilde{e}_1 \times_{\mathcal{B}} \widetilde{e}_2$, as our compiler is aware that $\times$ is more performant using an arithmetic circuit compared to a boolean one [18]. Rules (S-PCOND) and (S-SCOND) are along similar lines. Rule (S-PCOND) evaluates a public conditional to the sub-circuit from one of the branches, while the rule (S-SCOND) evaluates to a multiplexer mux gate that takes as input the sub-circuits from the guard (g_1) and both the branches (g_2 and g_3). Recall, for performance reasons, the expressions in the intermediate language generated by our compiler do not have $e_1\ ?_{\mathcal{A}}\ e_2\ :\ e_3$. Rules (S-COERCE) and (S-INP) evaluate to coerce and input gates respectively.

Statement evaluation $\widetilde{\rho} \vdash \widetilde{s} \Downarrow \widetilde{\rho}_1; \chi$ evaluates a statement $\widetilde{s}$ under the environment $\widetilde{\rho}$ to produce a new environment $\widetilde{\rho}_1$, and a circuit χ. Rules (S-DECLC), (S-DECLCKT), and (S-DECLCKTA) show the variable declaration cases. Rule (S-DECLC) shows the case when $\widetilde{e}$ evaluates to $\widetilde{v}$, where $\widetilde{v}$ is either a constant or an array of constants. In this case, the mapping $x \mapsto \widetilde{v}$ is added to the environment, and the resulting circuit is empty. When $\widetilde{e}$ evaluates to a sub-circuit g, rule (S-DECLCKT) picks a fresh wire w, adds the mapping $x \mapsto w$ to the environment $\widetilde{\rho}$, and outputs the circuit $\texttt{bind}\ g\ w$. The rule (S-DECLCKTA) is analogous for $\widetilde{e}$ evaluating to an array of sub-circuits. The variable assignment rules (not shown in the figure) are similar. The rule (S-WRITECKT) shows the case for writing to an array, where the array contents are secret. Finally, rule (S-OUT) compiles to an out circuit.

Circuit semantics. Evaluating a program in the intermediate language produces a circuit to be computed using a distributed 2PC protocol. With our circuit semantics, we model the *functional* aspect of a 2PC protocol, parametrized by cryptographic encoding and decoding functions.

During the circuit evaluation, the wire ids w are mapped to (random) strings b. The semantics of these strings is given by pairs of encode-decode algorithms, written as $\mathcal{E}_m$ and $\mathcal{D}_m$ (where m is either $\mathcal{A}$ or $\mathcal{B}$). More concretely, $\mathcal{E}_m(c)$ returns a pair of strings (b_1, b_2) with the property that $\mathcal{D}_m(b_1, b_2) = c$. The string b_j denotes the j^{th} party's *share* of c. We assume that the underlying 2PC protocol instantiates $\mathcal{E}_m$ and $\mathcal{D}_m$ appropriately. For the ABY protocol [18], algorithms $(\mathcal{E}_{\mathcal{A}}, \mathcal{D}_{\mathcal{A}})$ (resp. $(\mathcal{E}_{\mathcal{B}}, \mathcal{D}_{\mathcal{B}})$) correspond to the arithmetic (resp. boolean) secret-sharing and reconstruction algorithms.

$\boxed{\widehat{\rho}_1, \widehat{\rho}_2 \vdash g \Downarrow b_1, b_2}$ $\boxed{\widehat{\rho}_1, \widehat{\rho}_2 \vdash \chi \Downarrow \widehat{\rho}'_1, \widehat{\rho}'_2; O}$

C-IN
$$\frac{(b_1, b_2) = \mathcal{E}_m(c)}{\widehat{\rho}_1, \widehat{\rho}_2 \vdash \texttt{in}_j^m \Downarrow b_1, b_2}$$

C-COERCE
$$\frac{\widehat{\rho}_1, \widehat{\rho}_2 \vdash g \Downarrow b_1, b_2 \quad (b'_1, b'_2) = \mathcal{E}_m(\mathcal{D}_{m_1}(b_1, b_2))}{\widehat{\rho}_1, \widehat{\rho}_2 \vdash \langle m_1 \triangleright m \rangle\ g \Downarrow b'_1, b'_2}$$

C-MULT
$$\frac{\forall i \in \{1,2\}.\ \widehat{\rho}_1, \widehat{\rho}_2 \vdash g_i \Downarrow b_{1i}, b_{2i} \quad n_i = \mathcal{D}_{\mathcal{A}}(b_{1i}, b_{2i}) \quad (b_1, b_2) = \mathcal{E}_{\mathcal{A}}(n_1 \times n_2)}{\widehat{\rho}_1, \widehat{\rho}_2 \vdash \texttt{mult}\ g_1\ g_2 \Downarrow b_1, b_2}$$

C-GT
$$\frac{\forall i \in \{1,2\}.\ \widehat{\rho}_1, \widehat{\rho}_2 \vdash g_i \Downarrow b_{1i}, b_{2i} \quad n_i = \mathcal{D}_{\mathcal{B}}(b_{1i}, b_{2i}) \quad (b_1, b_2) = \mathcal{E}_{\mathcal{B}}(n_1 > n_2)}{\widehat{\rho}_1, \widehat{\rho}_2 \vdash \texttt{gt}\ g_1\ g_2 \Downarrow b_1, b_2}$$

C-MUX
$$\frac{\forall i \in \{1,2,3\}.\ \widehat{\rho}_1, \widehat{\rho}_2 \vdash g_i \Downarrow b_{1i}, b_{2i} \quad c_i = \mathcal{D}_{\mathcal{B}}(b_{1i}, b_{2i}) \quad (c_1 = \top) \Rightarrow ((b_1, b_2) = \mathcal{E}_{\mathcal{B}}(c_2)) \quad (c_1 = \bot) \Rightarrow ((b_1, b_2) = \mathcal{E}_{\mathcal{B}}(c_3))}{\widehat{\rho}_1, \widehat{\rho}_3 \vdash \texttt{mux}\ g_1\ g_2\ g_3 \Downarrow b_1, b_2}$$

C-BIND
$$\frac{\widehat{\rho}_1, \widehat{\rho}_2 \vdash g \Downarrow b_1, b_2 \quad \widehat{\rho}'_1 = \widehat{\rho}_1[w \mapsto b_1] \quad \widehat{\rho}'_2 = \widehat{\rho}_2[w \mapsto b_2]}{\widehat{\rho}_1, \widehat{\rho}_2 \vdash \texttt{bind}\ g\ w \Downarrow \widehat{\rho}'_1, \widehat{\rho}'_2; \cdot}$$

C-OUT
$$\frac{\widehat{\rho}_1, \widehat{\rho}_2 \vdash g \Downarrow b_1, b_2 \quad c = \mathcal{D}_m(b_1, b_2)}{\widehat{\rho}_1, \widehat{\rho}_2 \vdash \texttt{out}\ g \Downarrow \widehat{\rho}_1, \widehat{\rho}_2; c}$$

Fig. 10: Circuit semantics in a distributed runtime

Figure 10 gives the judgments for evaluation of circuits by the two parties using a 2PC protocol. The circuit environment is a map from wires to shares:

$$\text{Circuit environment} \quad \widehat{\rho} \quad ::= \quad \cdot \mid \widehat{\rho}, w \mapsto b$$

We use $\widehat{\rho}_j$ to denote the circuit environment for party j. We give the judgments $\widehat{\rho}_1, \widehat{\rho}_2 \vdash g \Downarrow b_1, b_2$, and $\widehat{\rho}_1, \widehat{\rho}_2 \vdash \chi \Downarrow \widehat{\rho}'_1, \widehat{\rho}'_2; O$, where O are the observations (similar to source semantics). The former judgment evaluates a gate under the environments $\widehat{\rho}_j$ and generates shares b_j of the gate's output. Rule (C-IN) evaluates the input gate $\texttt{in}_j^m$, and creates the m-type shares of the value c input by party j. Rule (C-MULT) illustrates the pattern for evaluating circuit gates g. To evaluate $\texttt{mult}\ g_1\ g_2$, the rule first evaluates g_1 to (b_{11}, b_{21}) and g_2 to (b_{12}, b_{22}). Shares (b_{11}, b_{21}) are then combined using $\mathcal{D}_{\mathcal{A}}$ to n_1, and similarly (b_{12}, b_{22}) are combined to n_2. The final output of the mult gate is then $\mathcal{E}_{\mathcal{A}}(n_1 \times n_2)$. Note that this is a functional description of how the mult gate evaluates, of course, concretely n_1 and n_2 are not observed by the parties. Rule (C-COERCE) creates the new shares using $\mathcal{E}_m$ (the corresponding rule for coercion from $\mathcal{P}$ is similar). The evaluation of bind updates the mapping of w in the input environments, and the rule (C-OUT) outputs the clear value c to the observations.

Correctness Theorem. We prove that all well typed programs always terminate successfully (array indices are always in bounds, there are no unbounded loops, etc.) and the 2PC protocol produces the same outputs as the source program. That is, if a source statement s is well-typed, and compiles to $\widetilde{s}$ in the intermediate language, then s terminates in the source semantics with observations O, $\widetilde{s}$ evaluates to circuit

χ, and χ terminates in the circuit semantics with the same observations O. Formally, the correctness theorem is as follows (the environments on the left of $\vdash$ are empty and we elide the environments on the right of $\vdash$ for brevity):

Theorem 1 (Correctness): $\forall\, s, \widetilde{s}$, if $\vdash s \rightsquigarrow \widetilde{s} \mid \_$ then $\exists O, \chi$, s.t. $\vdash s \Downarrow \_; O$, $\vdash \widetilde{s} \Downarrow \_; \chi$, and $\vdash \chi \Downarrow \_, \_; O$.

The proof follows by induction on the compilation derivation.

Security theorem. The protocols we generate provide simulation-based security against a semi-honest adversary, in the framework of [14], [38], [39] and provide provable security against all side-channel attacks. At a very high level, in this framework, parties are modeled as non-uniform interactive Turing machines (ITMs), with inputs provided by an environment $\mathcal{Z}$. An adversary $\mathcal{A}$, selects and "corrupts" one of the parties - however, $\mathcal{A}$ still follows the protocol specification. $\mathcal{A}$ interacts with $\mathcal{Z}$ that observes the view of the corrupted party. At the end of the interaction, $\mathcal{Z}$ outputs a single bit based on the output of the honest party and the view of the adversary. Two different interactions are defined: the *real world* and an *ideal world*. In the real interaction, the parties run the protocol Π in the presence of $\mathcal{A}$ and $\mathcal{Z}$. Let $\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}$ denote the distribution ensemble describing $\mathcal{Z}$'s output in this interaction. In the ideal interaction, parties send their inputs to a trusted functionality that performs the desired computation truthfully. Let $\mathcal{S}$ (the simulator) denote the adversary in this ideal execution, and $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ the distribution ensemble describing $\mathcal{Z}$'s output after interacting with the ideal adversary $\mathcal{S}$. A protocol Π is said to *securely realize* a functionality $\mathcal{F}$ if for every adversary $\mathcal{A}$ in the real interaction, there is an adversary $\mathcal{S}$ in the ideal interaction, such that no environment $\mathcal{Z}$, on any input, can tell the real interaction apart from the ideal interaction, except with negligible probability (in the security parameter κ). More precisely, the above two distribution ensembles are computationally indistinguishable.

We shall assume a cryptographic 2PC backend that securely implements any circuit χ that is output by our compiler (see Figure 8). This means that for any well-typed source program s, let χ be the circuit generated as in Theorem 1. We assume that there exists a 2PC protocol Π that securely realizes the functionality χ and let $\mathcal{S}_{2pc}$ be the corresponding simulator (that runs on χ, input of the corrupt party and the output obtained from trusted functionality for χ). We note that ABY [18] provides such a protocol Π and simulator $\mathcal{S}_{2pc}$ for all circuits χ output in our framework. We now state and prove our security theorem.

Theorem 2 (Security): Let s be a well typed program in our source language that generates a circuit χ (as defined in Theorem 1). Let protocol Π be the two-party secure computation protocol that securely realizes χ (as defined above). Then, Π securely realizes s.

Proof. Our simulator $\mathcal{S}$ simply runs our compiler on program s to obtain χ. It is crucial that this compilation to circuits does not require the secret inputs of the parties. Next, $\mathcal{S}$ sends the input of the corrupt party to the trusted functionality of s to obtain outputs O_1. Note that O_1 is same as the observations in the source semantics. By Theorem 1, these outputs O_1 are identical to outputs (or observations) O_2 of χ under circuit semantics. Next, $\mathcal{S}$ runs $\mathcal{S}_{2pc}$ on χ, input of the corrupt party and O_2. From the security of Π, we have that the simulated view output by $\mathcal{S}_{2pc}$ is indistinguishable from the real view. Hence, the security follows.

A. Additional Features

Though we model only one dimensional arrays above, our implementation supports multi-dimensional arrays as well. EzPC programs can have the following operators: addition, subtraction, multiplication, division by powers of two, left shift, logical and arithmetic right shifts, bitwise-(and, or, xor), unary negation, bitwise-negation, logical-(not, and, or, xor), and comparisons (less than, greater than, equality). Because of their high cost, integral division and floating-point operators are not supported natively by EzPC. However, we have implemented integral division in 30 lines of EzPC, while the floating-point support in ABY is under active development [40].

The formalization above restricts the array indices to be public and secret predicates are allowed only in multiplexers, i.e., in the `?:` operator and not in "`if`" statements. These restrictions can be removed by well-known *desugaring* transformations. Details of desugaring arrays with secret indices to arrays with public indices and multiplexers can be found in [9]. Similarly, see [41] for desugaring "`if`" statement with secret predicates to multiplexers. EzPC also has statements $\texttt{out}_1\ e$ (resp., $\texttt{out}_2\ e$) to reveal value of e to only the first (resp., second) party. Public inputs, i.e., inputs that are public to both parties are supported in two ways: either they can be hardcoded in the program text or they can be supplied by the parties to their individual protocol implementations at runtime[6] using the `in` keyword. To summarize, although we have focused on EzPC's features that are relevant to machine learning, we believe EzPC is useful for other applications of 2PC as well.

V. Secure code pipelining

In this section, we describe our "secure code pipelining" (SCP) technique that allows EzPC to execute programs that require large circuits. The reason to implement SCP is as follows: when compiling from a high level functionality to a circuit, the generated circuits, especially for ML applications, can quite often be larger than the total memory available on the machines executing the 2PC protocol. The secure protocol for evaluating this circuit cannot then be generated without swapping the circuit in and out of the main memory, thereby causing large slowdowns (see Section VII-B). To avoid this problem, we use SCP to "split" the high level functionality into smaller programs that compile into small (mixed arithmetic and boolean) circuits that can completely fit into the main memory. These small circuits are then evaluated sequentially, with proper threading of the intermediate outputs, to evaluate

[6]Parties provide consistent inputs as we assume semi-honest adversaries which follow the protocol description.

the entire program. Naturally, care must be taken to ensure that the partial evaluation of these circuits do not reveal any more information than the final output of the overall functionality.

Concretely, let s be a program in our source language that generates a circuit χ. For some programs, the circuit χ can be larger than the memory size[7] and fail to execute. SCP decomposes the program s into a sequence of smaller EzPC programs $t_1, t_2, \ldots, t_k$, as described below, such that the circuit size requirement for each of the t_i itself is manageable. We compile and execute each t_i sequentially, feeding the outputs of t_i as state information to t_{i+1}. We prove SCP to be correct (s and sequential execution of $t_1, t_2, \ldots, t_k$ compute the same functionality) and secure (sequential execution of $t_1, t_2, \ldots, t_k$ does not reveal any more information than s).

Our techniques take inspiration from the idea of pipelining Yao's garbled circuits described in FastGC [37]. However, unlike FastGC, we do not operate at a circuit level and the scheme itself is independent of the specific 2PC protocol. We now describe the instrumentation to ensure the security of SCP.

Let s takes (secret) inputs x from Alice and y from Bob and produces an output z to both parties. Let $s_1||s_2||\ldots||s_k$ be a decomposition of s such that the following holds. Define $q_0 = \perp$ (the public empty state). For all $1 \leq i \leq k-1$, s_i takes inputs x, y and q_{i-1} and outputs state q_i. Finally, s_k takes inputs x, y and q_{k-1} to output z. It is possible to decompose any program s into such $s_1||s_2||\ldots||s_k$. If EzPC generates circuit χ_i from s_i, the parties can execute $\chi_1, \chi_2, \ldots \chi_k$ sequentially (in a distributed setting) to obtain $q_1, \ldots, q_{k-1}$, and finally output z. At the i^{th} step, the parties only need to store information proportional to x, y, q_{i-1} and χ_i (which is much smaller than χ). However, this execution enables the parties to learn q_i (for all $1 \leq i \leq k-1$), which completely breaks the security.

To overcome this problem, we define a sequence of new programs t_i ($1 \leq i \leq k$) as follows. Once again, define $q_0 = \perp$. Without loss of generality, let all q_i be values in some additive ring $(\mathbb{Z}, +)$ (e.g., the additive ring $(\mathbb{Z}_{2^{64}}, +)$, i.e., the additive ring of integers modulo 2^{64}). Let $r_1, \cdots, r_{k-1}$ be a sequence of random values sampled from the same ring $(\mathbb{Z}, +)$ by Alice (in our implementation, all r_i values are generated by a pseudorandom function). Let t_1 be the program that takes as input x, r_1 from Alice and y from Bob (and empty state q_0), and runs s_1 (as defined above) to compute q_1 and then outputs $o_1 = q_1 + r_1$ *only to* Bob[8]. Alice's output from t_1 is r_1. Next, every t_i ($2 \leq i \leq k-1$) takes as inputs x, r_{i-1}, r_i from Alice and y, o_{i-1} from Bob, runs s_i on inputs x, y and state $q_{i-1} = (o_{i-1} - r_{i-1})$ (where $-$ denotes subtraction in the ring $(\mathbb{Z}, +)$) and then outputs $q_i + r_i$ to Bob and r_i to Alice. The last program t_k takes inputs x, y, r_{k-1}, o_{k-1}, runs s_k on inputs x, y and state $q_{k-1} = (o_{k-1} - r_{k-1})$ and outputs z to both parties. Although we have used arithmetic sharing here, Boolean sharing can be used to achieve the same effect.

Thus, given a decomposition of s into $s_1||s_2||\ldots||s_k$, we can use the construction above to generate programs $t_1, t_2, \ldots, t_k$, that can be sequentially executed, using the unmodified underlying 2PC backend. We prove the following theorem for SCP:

Theorem 3 (Correctness and security of SCP): If $s_1||s_2||\ldots||s_k$ is a decomposition of a program s, then there exists a sequence of programs $t_1, t_2, \ldots, t_k$ and protocols $\Pi_1, \Pi_2, \ldots, \Pi_k$ such that for all i, Π_i securely realizes t_i and $\Pi = \Pi_1, \Pi_2, \ldots, \Pi_k$ securely realizes s.

Proof. Let $t_1, \ldots, t_k$ be the sequence of programs as defined above corresponding to the decomposition $s = s_1||s_2||\ldots||s_k$. For every $1 \leq i \leq k$, let Π_i be the 2PC protocol output by our framework for t_i. Our construction for programs t_i ensures that if s is well-typed, then for each $1 \leq i \leq k$, t_i is well-typed. By Theorem 2, Π_i, the 2PC protocol that evaluates the circuit generated by t_i, securely realizes t_i. That is, for every $1 \leq i \leq k-1$, the Π_i provides observations r_i to Alice and o_i to Bob. Protocol Π_k provides observation z to both Alice and Bob. Finally, since r_i and o_i ($1 \leq i \leq k-1$) are individually uniformly random (in $(\mathbb{Z}, +)$, outputs received by the adversary can be simulated given the final output z.

A. Implementing SCP.

EzPC decomposes the source program s into a sequence of small programs $s_1||\ldots||s_k$, and then appropriately instruments them to produce $t_1, \ldots t_k$, as detailed above. It then compiles and executes the k programs $t_1, t_2, \ldots, t_k$ sequentially, freeing up memory usage after execution of each t_i. Automating the decomposition step requires an analysis that can statically estimate the resource usage of a EzPC program. Resource analysis of high-level programs is a well-known hard problem [42] and we describe a heuristic analysis.

To build s_1, we consider the longest prefix of s whose computation *size* is below the threshold enforced by the available memory of the machine. If $s = s_1; s_r$ then we recurse on s_r to obtain $s_2, \ldots, s_k$. For a program u, to estimate $size(u)$, we need to consider three important cases: if $u \equiv u_1; u_2$ then $size(u) = size(u_1) + size(u_2)$; if $u \equiv \texttt{if}(e_1, u_1, u_2)$ then $size(u) = \max(size(u_1), size(u_2))$;if $u \equiv \texttt{for}\, i\, \texttt{in}\, [n_1, n_2]\, \texttt{do}\, u_1$ then $size(u) = (n_2 - n_1) size(u_1)$. If $(n_2 - n_1) size(u_1)$ is above the threshold then we replace u by

$$\texttt{for}\, i\, \texttt{in}\, [n_1, \frac{n_1+n_2}{2}]\, \texttt{do}\, u_1; \texttt{for}\, i\, \texttt{in}\, [\frac{n_1+n_2}{2}, n_2]\, \texttt{do}\, u_1$$

and recurse to find the prefix again. This heuristic analysis is sufficient for the benchmarks discussed in our evaluation.

B. Example

We now illustrate SCP through an example. Consider the functionality in Figure 11. This is a functionality that takes as input two vectors w and v from Alice and two vectors x and y from Bob. It computes two inner products $w^T x$ and $v^T y$, compares the first value with the second and returns a

[7] In fact, there is an upper limit of $2^{32} - 1$ gates for the circuit size in ABY but for most machines the memory limit is hit first.

[8] While the description of the scheme here assumes that the underlying backend supports only one party receiving output, this is only a simplifying assumption, and we can easily modify our protocol in the case where both parties must receive the same output.

```
uint w[30] = input1();  uint v[30] = input1();
uint x[30] = input2();  uint y[30] = input2();
uint acc1 = 0;  uint acc2 = 0;
for i in [0 : 30]
{acc1 = acc1 + (w[i] × x[i]);
 acc2 = acc2 + (v[i] × y[i]);}
output2((acc1 > acc2) ? 1 : 0) //only to party 2
```

Fig. 11: EzPC code for $w^T x > v^T y$

```
uint w[30] = input1();  uint r1 = input1();
uint x[30] = input2();
uint acc1 = 0;
for i in [0 : 30] {acc1 = acc1 + (w[i] × x[i]);}
uint o1 = acc1 + r1;
output2(o1) //acc1 is ``secret shared''
```

Fig. 12: Program 1: Code for $o_1 = w^T x + r_1$

boolean value (which is 1 if $w^T x > v^T y$ and 0 otherwise) to Bob. Now, if we wish to split this functionality using SCP, one possible split is into the following three programs[9]. Program 1 (Figure 12) computes $w^T x$ and "secret shares" the output of this computation between Alice and Bob (Alice's share is r_1, a random value, and Bob's share is $o_1 = w^T x + r_1$). Next, program 2 (Figure 13) computes $v^T y$ and once again provides Alice with r_2 and Bob with $o_2 = v^T y + r_2$. Finally, program 3 (Figure 14) compares $o_1 - r_1$ with $o_2 - r_2$ and provides the output to Bob. It is easy to see that the size of the programs $1, 2$ and 3 (and their corresponding circuits output by the EzPC compiler) are smaller than the program in Figure 11 and its corresponding circuit, and in particular, smaller than the state that must be maintained between the programs.

VI. Implementation

We discuss some implementation details of EzPC. The EzPC compiler compiles each of our benchmarks in under a second to C++ code that makes calls to the ABY library [18]. ABY provides support for Arithmetic computations based on [20], and boolean computations based on GMW [14] as well as Yao's garbled circuits [13]. Although EzPC can generate code for both kinds of boolean computations, we have observed better performance when using garbled circuits and use it in our evaluation. Hence, for our benchmarks, EzPC generated code uses arithmetic computations and garbled circuits based boolean computations. We use 128 bits of security and OT extension-based arithmetic multiplication triplets generation.

We use an off-the-shelf verification framework, SeaHorn [43], to check that the array indices are within bounds ($\Gamma \models e < n$ in (T-Read) and (T-Write), Figure 7). We take the EzPC source program and translate it as an input C program to SeaHorn, which then takes less than a minute on our largest benchmark to verify that all the array accesses are in-bounds.

Our implementation assigns the type labels (rule T-Decl) conservatively. Only the `for`-loop counters are assigned public

[9]All arithmetic is over an appropriate ring in the following discussion.

```
uint v[30] = input1();  uint r2 = input1();
uint y[30] = input2();
uint acc2 = 0;
for i in [0 : 30] {acc2 = acc2 + (v[i] × y[i]); }
uint o2 = acc2 + r2;
output2(o2) //acc2 is ``secret shared''
```

Fig. 13: Program 2: Code for $o_2 = v^T y + r_2$

```
uint r1 = input1();  uint r2 = input1();
uint o1 = input2();  uint o2 = input2();
uint acc3 = o1 − r1;  uint acc4 = o2 − r2;
output2((acc3 > acc4) ? 1 : 0) //only to party 2
```

Fig. 14: Program 3: Code for $(o_1 - r_1) > (o_2 - r_2)$

labels. All other variables are assigned arithmetic labels (that can later be coerced to boolean). While assigning public labels to loop counters is critical for performance and/or security, assigning arithmetic labels to other variables incurs minimal cost in practice. While a more sophisticated type inference procedure is certainly possible (e.g., based on network probing [24]), such approaches invariably increase the compilation time. We believe short compilation times are essential for a good programming experience; the EzPC compiler, with all its optimizations, has a sub-quadratic time complexity and takes less than a second in practice.

The compilation rules of Figure 7 can introduce repeated coercions from arithmetic to boolean and vice versa. Since EzPC is aware of the cryptographic costs associated with these coercions, it tries to minimize them using several optimizations, e.g., by the standard "common subexpression elimination" optimization [44]. On each coercion, EzPC memorizes the pair of arithmetic share and boolean share involved in the coercion. EzPC invalidates such pairs when the variables corresponding to the shares are overwritten by assignments. In subsequent coercions, EzPC reuses valid pairs (if available) instead of inserting code to recompute them afresh. These optimizations are standard compiler optimizations [44], and we rely on their correctness (optimizations preserve outputs and well-typedness) for the security of the optimized programs.

Our compiler also provides a C++ backend that compiles the source EzPC programs to plain C++ programs, without using any cryptography. This backend, along with the usual ABY backend, can be used for lightweight differential testing of the functionality itself [45].

VII. Evaluation

We evaluate EzPC on a variety of problems that can fall under the umbrella of *secure prediction*, where one party (the server) has a machine learning model, and the other party (the client) has an input. The goal is to compute the output of the model on client's input, with the guarantee that the server learns nothing about the input, and the client learns nothing about the model beyond what is revealed from the output.

Dataset	d	Prev time (s)	Prev comm (KB)	LAN (s)	WAN (s)	Comm (KB)	Num gates	LOC
Breast cancer	30	0.3	36	0.1	0.3	25	727	20
Credit	47	0.3	41	0.1	0.3	36	795	20

TABLE I: Linear classification results. We compare with [1] .

Dataset	n	F	Prev time (s)	Prev comm (MB)	LAN (s)	WAN (s)	Comm (MB)	Num gates	LOC
Nursery	5	9	1.5	0.2	0.1	0.4	0.6	73k	50
Audiology	24	70	3.9	2.0	1.5	2.9	37	4219k	50

TABLE II: Naïve Bayes results. We compare with [1].

Dataset	d	N	Prev time (s)	Prev comm (KB)	LAN (s)	WAN (s)	Comm (KB)	Num gates	LOC
Nursery	4	4	0.3	102	0.1	0.3	32	3324	20
ECG	4	6	0.4	102	0.1	0.4	49	5002	20

TABLE III: Decision tree benchmarks. We compare with [2].

To begin, we first implement the benchmarks from Bost et al. [1] and MINIONN [4] (both of which study the same setting), and show that the performance of the high-level code written in EZPC is comparable to their hand-crafted protocols. Next, we demonstrate the generality and programmability aspects of EZPC by implementing state-of-the-art machine learning models from Tensorflow [27] and BONSAI [28]. Indeed, we provide the first 2PC implementation of BONSAI. We implement a Deep Neural Network (DNN) for CIFAR-10 dataset [46] from MINIONN [4] and matrix factorization [25] to evaluate SCP. We focus on comparing with hand-crafted protocols as the prior frameworks for secure computation incur more than 25x slowdown on these tasks (Section I-D).

We present the numbers for two network settings, a LAN setting and a cross-continent WAN setting. The round trip time between the server and the client machines in the two settings is 1ms and 40ms respectively. When we compare our execution times with prior protocols, we match our system and network parameters with those of the prior work. Most of our benchmarks are related to machine learning and we refer the reader to the appropriate papers for descriptions of the algorithms.

A. Secure prediction

Standard classifiers. We evaluate the three standard classifiers, linear, Naïve Bayes, and decision trees, from [1] on the following data sets from the UCI machine learning repository [47]: the Wisconsin Breast Cancer data set, Credit Approval data set, Audiology (Standardized) data set, Nursery data set, and ECG (electrocardiogram) classification data from [3].

The results for linear classification are in Table I. The input and the model are both vectors of length d. The columns "Prev. time" and "Prev. comm" show the time and the total network communication reported by Bost et al. [1] for a network setting with 40ms round trip time, which is same as our WAN setting. The total execution time of EZPC generated code in the LAN and the WAN setting is reported next, followed by the total communication. We observe that the EZPC code performance matches the hand-crafted protocol of Bost et al., and the programmer effort in EZPC is just 20 lines (last column in the table) of high-level code in the EZPC source language.

The results for Naïve Bayes are in Table II. As before, n denotes the number of classes and F is the number of features. As before, we compare with Bost et al. [1] and observe that EZPC generated code has better performance, despite using a generic 2PC, as opposed to custom designed protocols developed by Bost et al. Moreover, they remark that in their setup, generic Yao-based 2PC did not scale to the smallest of their Naïve Bayes classifiers, so they had to scale down the prediction task, and even then Yao-based 2PC was 500x slower. Whereas, we show that by using a cryptographic cost aware compiler, we can scale generic 2PC to real prediction tasks, and get performance competitive to or better than the specialized protocols. Table III compares against the more recent work of [2] on decision trees and further validates this claim.

Deep neural nets. We evaluate EZPC on the DNNs described in SecureML [5], Cryptonets [26], and the CNN from MINIONN [4]. For comparison, we consider their implementations from MINIONN [4], which outperforms their previous implementations. Table IV shows the results. We note that for each of these DNNs, MINIONN provides a specialized protocol, while EZPC uses a generic 2PC protocol (auto) generated from high-level code.

The first benchmark is the DNN described in SecureML [5] (Figure 10 in [4]). It has three fully connected layers with square as the activation function. Next, we implement the DNN described in Cryptonets [26] (Figure 11 in [4]) in EZPC. This DNN also uses square as the activation function and has one convolution (with 5 output channels) and one fully connected layer. Finally, we implement CNN from MINIONN (Figure 12 in [4]), that has two convolutions (with 16 output channels each) and two fully connected layers. In contrast to the previous two DNNs, it uses ReLU for activation and has significantly higher number of boolean-and gates. Note that square activation can be implemented entirely using arithmetic gates but ReLU requires boolean-and gates. For a complete description of these benchmarks and their accuracies, we refer the reader to the original references.

DNN	Prev time (s)	Prev comm (MB)	LAN (s)	WAN (s)	Comm (MB)	Num gates	Model size	LOC
SecureML	1.1	15.8	0.7	1.7	76	366k	119k	78
Cryptonets	1.3	47.6	0.6	1.6	70	316k	86k	88
CNN	9.4	657.5	5.1	11.6	501	9480k	35k	154

TABLE IV: DNN benchmarks. We compare with [4].

Classifier	LAN (s)	WAN (s)	Comm (MB)	Num And	Num Mul	Num gates	Model size	LOC
Regression	0.1	0.7	5	2k	8k	35k	8k	38
CNN	30.5	60.3	2955	6082k	4163k	42104k	3226k	172

TABLE V: Tensorflow tutorial benchmarks

Dataset	LAN (s)	WAN (s)	Comm (MB)	Num And	Num Mul	Num gates	depth	LOC
Chars4k	0.1	0.7	2	18k	3k	85k	1	89
USPS	0.2	0.9	4	62k	2k	285k	2	156
WARD	0.3	1.1	9	106k	8k	506k	3	283

TABLE VI: Bonsai benchmarks

In Table IV, the column "Model size" is the number of parameters in the trained model. We observe that our performance is competitive with specialized MINIONN protocols, for both the LAN and the WAN settings. Further, lines of EZPC source code required is still small. We note that while the MINIONN implementation is based on the ABY framework, it does not use ABY "off-the-shelf" and performs application-specific optimizations. In contrast, EZPC focuses on generic 2PC and directly exploits the existing performant implementations in ABY. MINIONN also reports performance results on a bigger DNN with 7 convolution layers. In EZPC, this benchmark requires SCP and we discuss it in Section VII-B.

State-of-the-art classifiers. Tensorflow [27] is a standard machine learning toolkit. Its introductory tutorial describes two prediction models for handwritten digit recognition using the MNIST dataset [48]. Each image in this dataset is a greyscale 28×28 image of digits 0 to 9. The first model that the tutorial describes is a softmax regression that provides an accuracy of 92%. The classifier evaluates $\mathsf{argmax}\ W \cdot x + b$. Here, x is a 784 length vector obtained from the input image, W is a 10×784 matrix, and b is a 10 length vector. We implement this classifier in EZPC and present the results in the first row of Table V.

The next classifier in the Tensorflow tutorial is a convolution neural net with two convolutions (with 32 output channels) and two fully connected layers with ReLU as the activation function. This DNN is both bigger and more accurate than the DNNs presented in the previous section. In particular, it has an accuracy of 99.2%. Since, we are not aware of any other tools that have used this model as a benchmark, we only report numbers for EZPC. We observe that this DNN can take a minute per prediction in the WAN setting and is the largest benchmark that we have evaluated without SCP.

We next present BONSAI [28] results on three standard datasets: character recognition (Chars4k [49], accuracy 74.71%), text recognition (USPS [50], accuracy 94.4%), and object categorization (WARD [51], accuracy 95.7%). BONSAI takes as input $x \in \mathbb{R}^d$, and its model consists of a binary tree with N nodes, and a matrix Z. Each node j contains matrices W_j and V_j, and a vector θ_j. The internal node j evaluates a predicate $(\theta_j^T \cdot Z \cdot x) > 0$ to decide whether to pass x to the left child $2j+1$ or the right child $2j+2$. The predicted value is

$$\mathsf{argmax} \sum_{j=0}^{N-1} I_j(x)[(W_j^T \cdot Z \cdot x) \circ (f(V_j^T \cdot Z \cdot x))]$$

Here, $I_j(x)$ is 1 if the j^{th} node is present on the path traversed by x and is zero otherwise. The operation $\circ$ is a pointwise multiplication of two vectors, W_j's and V_j's are matrices of appropriate dimensions. The activation function f is given by $f(y) = y$ if $-1 < y < 1$ and $\mathrm{sign}(y)$ otherwise.

We implement the trained classifiers in EZPC for all the benchmarks from [28], and show the representative results in Table VI. Out of all the benchmarks from [28], the dataset WARD requires the largest model. The column "depth" shows the depth of the tree used by BONSAI.

To summarize, by providing first 2PC implementations of state-of-the-art classifiers, we have demonstrated the expressiveness of EZPC. We discuss scalability next.

	LAN (s)	WAN (s)	Comm (MB)	Num And	Num Mul	Num gates
Total	265.6	647.5	40683	21m	61m	337m
Stage 6	55.2	122.6	6744	12m	10m	98m

TABLE VII: SCP results for CIFAR-10. MINIONN takes 544 seconds and communicates 9272 Mb.

Stage	LAN (s)	WAN (s)	Comm (MB)	depth	Num gates	LOC
1	175	662	29816	16370	33m	500
2	193	1095	31945	30916	37m	516
3	178	627	29810	16369	32m	478
Total	546	2384	91571	–	102m	1494

TABLE VIII: SCP results for matrix factorization. The time reported by [25] for this computation is 10440 seconds.

B. SCP evaluation

The largest benchmark of MINIONN [4] is a DNN for CIFAR-10 dataset [46]. The classifier's task is to categorize colored (32×32) images into 10 classes. A secure evaluation of this DNN needs more memory than what is available on our machines. Therefore, we use SCP and divide the computation into seven stages. The first step does a convolution with 64 output channels and a ReLU activation. The next four stages together perform a convolution that involves multiplying a 64×576 matrix with a 576×1024 matrix. The sixth stage performs a ReLU and a convolution. The final stage has four convolutions, five ReLUs, and a fully connected layer. The total number of lines of EZPC code for this benchmark is 336 lines.

Table VII shows the end-to-end numbers as well as the numbers for the sixth stage, which is the heaviest. The number of gates are in millions, hence the suffix 'm' in the last three columns. As with Table IV, EZPC generated generic 2PC protocol is competitive with MINIONN here as well. Therefore, we believe that with SCP, EZPC can scale to large

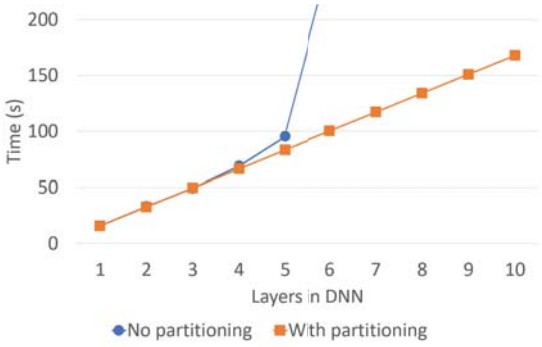

Fig. 15: Comparison of EzPC code with and without SCP. x-axis denotes the number of layers in the DNN, while y-axis denotes time in seconds for the secure protocol.

computations while maintaining performance competitive with existing specialized protocols. In particular, for a large enough DNN, MINIONN could run out of memory but an appropriately pipelined EzPC implementation would still succeed.

Scalability. To illustrate the scalability of SCP, we evaluate a sequence of DNNs with and without SCP in Figure 15. All layers are identical and SCP places each layer in a separate stage. For DNNs with up to 4 layers, the performance, with and without SCP, is almost identical and the lines overlap, thereby illustrating that SCP does not cause any noticeable performance overheads. Memory issues start showing up in the non-pipelined implementation of the 5 layer DNN and it is slower. In particular, the circuit for this DNN does not fit in RAM and uses swap space. Performance degrades rapidly thereafter: executing the 5 layer DNN and half operations of the sixth layer incurs a 2x slowdown. DNNs with 6 or more layers fail to execute (terminate with a "bus error"). However, the pipelined implementation scales well to even these large DNNs.

C. Matrix factorization

EzPC is not tied to secure prediction and can express more general computations. To demonstrate this expressiveness, we implement secure matrix factorization [25]. Abstractly, given a sparse matrix $\mathcal{M}$ of dimensions $n \times m$ and M non-zero entries, the goal is to generate a matrix U of dimension $n \times d$ and a matrix V of dimension $d \times m$ such that $\mathcal{M} \approx UV$. This operator is useful in recommender systems. In particular, Nikolaenko et al. [25] shows how to implement a movie recommender system which does not require users to reveal their data in the clear, i.e., the ratings the users have assigned to movies are kept secret. The implementation is a two party computation of an iterative algorithm for matrix factorization (Algorithm 1 in [25]). This algorithm is based on gradient descent and iteratively converges to a local minima. We implement this algorithm in EzPC.

To ensure that the algorithm converges to the right local minima, Nikolaenko et al. require 36 bits of precision. Since ABY supports either 32-bit or 64-bit integers, our EzPC implementation manipulates 64-bit variables. For the matrix $\mathcal{M}$ of user data, Nikolaenko et al. consider $n = 940$ users, $m = 40$ most popular movies, and $M = 14683$ ratings from the MovieLens dataset. The time reported in [25] for one iteration is 2.9 hours. This computation is large enough that we pipeline each iteration into three stages. The first stage involves a Batcher [52] sorting network followed by a linear pass. The second stage involves sorting and gradient computations and is the heaviest stage. The third stage is similar to the first stage. The results are presented in Table VIII. These circuits have a large depth (column "depth"); the circuits for secure prediction had depth below 100.

We observe that in the LAN setting, we are about 19 times faster than [25] The main source of this significant speedup is that, unlike [25], EzPC does not need to convert the functionality into boolean circuits. However, this benchmark requires more lines of code than the previous benchmarks because of Batcher's sort (450 lines of EzPC code in each stage). However, the current programmer effort seems minuscule compared to the mammoth implementation effort put in by Nikolaenko et al. (Section 5 of [25]) to scale a boolean circuits based backend to this benchmark.

VIII. Related work

EzPC falls into the category of frameworks that compile high level languages to 2PC protocols. We discuss other such frameworks next. Fairplay's Secure Function Definition Language (SFDL) [6], [53] and CBMC-GC [9] compile C or Pascal like programs into boolean circuits that are then evaluated using garbled circuits [13]. ObliVM [8] protects access patterns using an oblivious RAM [54], [55] and also uses garbled circuits for compute. In Secure Multiparty Computation Language (SMCL) [10], Java like programs are compiled into arithmetic circuits that are then evaluated using the VIFF framework [16]. Wysteria [7] enables programmers to write n-party mixed-mode programs that combine local, per-party computations with secure computations. It compiles secure computations to boolean circuits and uses a GMW-based backend [56], [14]. Mitchell et al. [12] allow the user to select between Shamir's secret sharing [15] and fully homomorphic encryption [17], while Alchemy [57] provides compilation of plaintext code into fully homomorphic encryption. The work of Araki et al. [58] generalizes the SPDZ compiler to multiple backend protocols for multi-party computation; however, mixing of protocols is still not supported. Unlike EzPC, all these tools use either an arithmetic backend or a boolean backend but not a combination of both.

Next, we discuss tools that expose libraries which developers can use to describe 2PC protocols. To generate efficient protocols for a functionality, the programmer must break the functionality into components and call the appropriate library functions. For example, ABY [18] falls in this category. The TASTY tool [19] allows mixing homomorphic encryption based arithmetic computations and garbled circuits based boolean computations and the interconversions between the two are inserted by the programmer explicitly. Kerschbaum

et al. [20] and Pattuk et al. [24] provide schemes to automatically assign homomorphic encryption or garbled circuits to each operator in a computation. They operate over circuits represented as a sequence of dyadic operations, while EzPC provides a high-level programming language with loops and branches. EzPC compilation is automatic and sub-quadratic in the size of generated ABY programs (e.g. Figure 3) after switching off memory safety checking. In sharp contrast, [20] and [24] describe approaches that would be exponential in the length of ABY programs. As a result, these techniques are not directly applicable to EzPC. Other examples include the VIFF framework [16] for arithmetic computations and Sharemind [11] (secure 3-party boolean computation).

2PC backends have made tremendous progress in the last decade. For example, the circuits can be optimized for depth [40], [59], large garbled circuits can be pipelined [37], [8], online complexity can be reduced at the cost of offline complexity [60], encrypted values output from a garbled circuit can be reused [61] and oblivious RAM [54], [55] can be used to hide access patterns of MIPS code [62]. ABY^3 [63] shows an ABY-like backend for 3-party secure computation protocols and could potentially serve as another backend to EzPC. Incorporating these backends would only improve the performance and scalability of EzPC implementations.

Many works have designed specialized protocols for various 2PC tasks. This requires deep knowledge of cryptography to ensure security. Examples include [3], [30], [31], [32], [33], [25], [34], [1], [2], [5], [4].

There have been several recent works in the area of executing secure machine learning tasks using 2PC. Juvekar et al. [64] presented GAZELLE, which provides new, specialized protocols for convolutional neural networks (CNNs) that are more efficient than ABY: GAZELLE can evaluate the CNN benchmark in Table IV in 0.8 seconds, as opposed to 5.1 seconds taken by EzPC with the ABY backend. In the future, once GAZELLE is available, we could add it as another cryptographic backend to EzPC and improve the productivity of GAZELLE's users.

HyCC [29] is a recent ABY-based system that translates an unspecified subset of ANSI-C to ABY without any formal guarantees about the translation process. HyCC does not scale to large neural networks like CIFAR-10 and has been evaluated only on smaller benchmarks whose circuits fit in memory. For the benchmarks that HyCC can handle, EzPC generates identical protocols to HyCC and the EzPC compiler is at least 600X faster than the HyCC system.

IX. Conclusion and Future Work

We presented EzPC, the first cryptographic cost aware framework that generates efficient and scalable 2PC protocols from high-level programs. The compiler is backed by formal semantics that help it maintain correctness, security, and efficiency. The generated protocols comprise combinations of arithmetic and boolean circuits and have performance comparable to, or better than the previously known custom specialized protocols from previous works. Because of its scalability and rich syntax, we believe EzPC can express arbitrary machine learning models that arise in practice. The only remaining bottleneck for such tasks is the communication and computational complexity of the state-of-the-art 2PC protocols.

Currently, we are working on a front-end to translate Tensorflow code to EzPC. The aim here is to provide a push button implementation that generates secure implementations for existing Tensorflow models. In the future, we would like to extend our security guarantees to malicious adversaries. We think that, with suitable changes to operator costs, EzPC can target other backends including 3PC or MPC, or protocols secure against malicious adversaries. The cryptographic backends continue to improve and the modular design of EzPC makes it easy to integrate with the best available backends. However, we are currently unaware of a maliciously secure 2PC implementation for combinations of arithmetic and boolean circuits. On the compiler side, we are exploring the possibility of mechanically verifying the compiler implementation. Finally, we would like to conduct user studies to quantify the impact of EzPC on programmer productivity.

References

[1] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser, "Machine learning classification over encrypted data," in *NDSS*, 2015.

[2] D. J. Wu, T. Feng, M. Naehrig, and K. E. Lauter, "Privately evaluating decision trees and random forests," *PoPETs*, vol. 2016, 2016.

[3] M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A. Sadeghi, and T. Schneider, "Secure evaluation of private linear branching programs with medical applications," in *ESORICS*, 2009.

[4] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *CCS*, 2017.

[5] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," in *IEEE S&P*, 2017.

[6] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay - secure two-party computation system," in *USENIX Security*, 2004.

[7] A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A programming language for generic, mixed-mode multiparty computations," in *IEEE S&P*, 2014.

[8] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *IEEE S&P*, 2015.

[9] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith, "Secure two-party computations in ANSI C," in *CCS*, 2012.

[10] J. D. Nielsen and M. I. Schwartzbach, "A domain-specific programming language for secure multiparty computation," in *PLAS*, 2007.

[11] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations," in *ESORICS*, 2008.

[12] J. C. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman, "Information-flow control for programming on encrypted data," in *IEEE CSF*, 2012.

[13] A. C. Yao, "How to generate and exchange secrets (extended abstract)," in *FOCS*, 1986.

[14] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game or A completeness theorem for protocols with honest majority," in *STOC*, 1987.

[15] R. Gennaro, M. O. Rabin, and T. Rabin, "Simplified VSS and fact-track multiparty computations with applications to threshold cryptography," in *PODC*, 1998.

[16] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen, "Asynchronous multiparty computation: Theory and implementation," in *PKC*, 2009.

[17] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *STOC*, 2009.

[18] D. Demmler, T. Schneider, and M. Zohner, "ABY - A framework for efficient mixed-protocol secure two-party computation," in *NDSS*, 2015.

[19] W. Henecka, S. Kögl, A. Sadeghi, T. Schneider, and I. Wehrenberg, "TASTY: tool for automating secure two-party computations," in *CCS*, 2010.

[20] F. Kerschbaum, T. Schneider, and A. Schröpfer, "Automatic protocol selection in secure two-party computations," in *ACNS*, 2014.

[21] 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 *ASIACCS*, 2018.

[22] D. Bogdanov, P. Laud, and J. Randmets, "Domain-polymorphic language for privacy-preserving applications," in *Proceedings of the First ACM Workshop on Language Support for Privacy-enhancing Technologies*, ser. PETShop '13, 2013.

[23] A. Schröpfer, F. Kerschbaum, and G. Müller, "L1 - an intermediate language for mixed-protocol secure computation," in *COMSAC*, 2011.

[24] E. Pattuk, M. Kantarcioglu, H. Ulusoy, and B. Malin, "CheapSMC: A framework to minimize secure multiparty computation cost in the cloud," in *DBSec*, 2016.

[25] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh, "Privacy-preserving matrix factorization," in *ACM CCS*, 2013.

[26] 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 *ICML*, 2016.

[27] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016.

[28] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 KB RAM for the internet of things," in *ICML*, 2017.

[29] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider, "Hycc: Compilation of hybrid protocols for practical secure computation," in *CCS 2018*.

[30] M. Blanton and P. Gasti, "Secure and efficient protocols for iris and fingerprint identification," in *ESORICS*, 2011.

[31] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel, "Privacy-preserving remote diagnostics," in *CCS*, 2007.

[32] M. Franz, B. Deiseroth, K. Hamacher, S. Jha, S. Katzenbeisser, and H. SchröDer, "Secure computations on non-integer values with applications to privacy-preserving sequence analysis," *Inf. Secur. Tech. Rep.*, vol. 17, no. 3.

[33] Y. Huang, L. Malka, D. Evans, and J. Katz, "Efficient privacy-preserving biometric identification," in *NDSS*, 2011.

[34] 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*, 2013.

[35] A. Schröpfer and F. Kerschbaum, "Forecasting run-times of secure two-party computation," in *QEST*, 2011.

[36] E. M. Songhori, M. S. Riazi, S. U. Hussain, A.-R. Sadeghi, and F. Koushanfar, "Arm2gc: High-level and reliable secure computation framework," IACR ePrint, 2017.

[37] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure two-party computation using garbled circuits," in *Usenix Security*, 2011.

[38] R. Canetti, "Security and composition of multiparty cryptographic protocols," *J. Cryptology*, vol. 13, no. 1, 2000.

[39] ——, "Universally composable security: A new paradigm for cryptographic protocols," in *FOCS*, 2001.

[46] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[40] D. Demmler, G. Dessouky, F. Koushanfar, A. Sadeghi, T. Schneider, and S. Zeitouni, "Automated synthesis of optimized circuits for secure computation," in *ACM CCS*, 2015.

[41] S. Zahur and D. Evans, "Obliv-c: A language for extensible data-oblivious computation," IACR ePrint, 2015.

[42] J. Hoffmann, A. Das, and S. Weng, "Towards automatic resource bound analysis for ocaml," in *SIGPLAN*, 2017.

[43] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *CAV*, 2015.

[44] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[45] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, 1998.

[47] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[48] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[49] T. E. de Campos, B. R. Babu, and M. Varma, "Character recognition in natural images," in *VISAPP*, 2009.

[50] J. J. Hull, "A database for handwritten text recognition research," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 16, no. 5, 1994.

[51] J. Yang, Y. Li, Y. Tian, L. Duan, and W. Gao, "Group-sensitive multiple kernel learning for object categorization," in *ICCV*, 2009.

[52] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring), 1968.

[53] A. Ben-David, N. Nisan, and B. Pinkas, "Fairplaymp: a system for secure multi-party computation," in *ACM CCS*, 2008.

[54] R. Ostrovsky, "Efficient computation on oblivious rams," in *STOC*, 1990.

[55] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, 1996.

[56] S. G. Choi, K. 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*, 2012.

[57] E. Crockett, C. Peikert, and C. Sharp, "ALCHEMY: A language and compiler for homomorphic encryption made easy," in *CCS 2018*.

[58] T. Araki, A. Barak, J. Furukawa, M. Keller, Y. Lindell, K. Ohara, and H. Tsuchida, "Generalizing the SPDZ compiler for other protocols," in *CCS 2018*.

[59] N. Büscher, A. Holzer, A. Weber, and S. Katzenbeisser, "Compiling low depth circuits for practical secure computation," in *ESORICS*, 2016.

[60] A. Groce, A. Ledger, A. J. Malozemoff, and A. Yerukhimovich, "Compgc: Efficient offline/online semi-honest two-party computation," *IACR Cryptology ePrint Archive*, 2016.

[61] B. Mood, D. Gupta, K. R. B. Butler, and J. Feigenbaum, "Reuse it or lose it: More efficient secure computation through reuse of encrypted values," *CoRR*, 2015. [Online]. Available: http://arxiv.org/abs/1506.02954

[62] X. S. Wang, S. D. Gordon, A. McIntosh, and J. Katz, "Secure computation of MIPS machine code," in *ESORICS*, 2016.

[63] P. Mohassel and P. Rindal, "Aby^3: A mixed protocol framework for machine learning," in *CCS 2018*.

[64] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasani, "GAZELLE: A low latency framework for secure neural network inference," in *USENIX Security 18*, 2018.