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

Nishanth Chandran
MSR India
nichandr@microsoft.com

Divya Gupta
MSR India
t-digu@microsoft.com

Aseem Rastogi
MSR India
aseemr@microsoft.com

Rahul Sharma
MSR India
rahsha@microsoft.com

Shardul Tripathi
IIT Delhi
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 sharing and garbled circuits for better performance. We empirically demonstrate that the protocols generated by our framework match or outperform (up to 19x) recent works that provide hand-crafted protocols for various functionalities such as secure prediction and matrix factorization.

1 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 ([3, 9, 40, 44, 56]), 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, $^T$ is the matrix transpose operator, and $w^T x$ denotes the inner product of $w^T$ and $x$. Alice has the following options.

1 Read as “easy peasy”, stands for Easy 2 Party Computation.
At a high level, we decompose the original program into a sequence of data.

The outcome is an efficient 2PC protocol combining arithmetic and boolean representations of the source language operators.

EzPC 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 inter-conversion 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.

Scalability (secure code partitioning). 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. This is a show-stopper for applications like secure machine learning, secure prediction, etc. that operate on large amounts of data. EzPC addresses the scalability concern using a novel technique that we call secure code partitioning (or partitioning 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 5). As we show in our evaluation, partitioning allows us to program large applications in EzPC.

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 adversaries to the semi-honest security of the 2PC back-end. 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 our partitioning scheme (Section 4 and Section 5).

Evaluation. We have implemented EzPC using ABY [18] as the cryptographic back-end. We compare EzPC with Yao-based compilers in Section 2.1 and with specialized protocols in Section 7. We evaluate EzPC by implementing a wide range of secure prediction benchmarks including linear and naive bayes classifiers, decision trees, deep neural networks, state-of-the-art classifiers from TensorFlow [1] and Bonsai [36], and also the matrix factorization example from Nikolaenko et al. [48]. Our results demonstrate three key points. First, EzPC makes it convenient for general programmers to write 2PC protocols. E.g. we provide the first 2PC implementation of Bonsai [36], and it was programmed in the high-level EzPC source language by a non-cryptographer. Second, the performance of the protocols generated by EzPC are comparable to or better than (up to 19x) their state-of-the-art, hand-crafted implementations. Finally, we demonstrate the usefulness of partitioning by implementing an application that requires more than 300 million gates (Section 6 and Section 7).

Related Work. Before ABY [18], several works have proposed combining secure computation protocols based on homomorphic encryption and Yao’s garbled circuits (e.g. [3, 6, 10, 19, 31, 48, 49, 53]), and some have also developed tools that allow writing such combinations (e.g. [7, 27, 34, 54]). 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 provide informal languages or libraries that lack formal semantics and static guarantees. Finally, we focus on the language features necessary to implement machine learning applications. In particular, we do not discuss declassification of intermediate values or indexing into arrays at secret indices. Handling them requires additional complexity. For example, Wysteria [51] handles the former using dependent types and OblivVM [39] uses Oblivious RAM for

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

Figure 1: EzPC code for $w^T x > b$
the latter. These situations seldomly arise in the machine learning applications that we consider: intermediate values don’t need to be declassified and arrays are traversed in an oblivious manner. We provide a detailed survey of related work in Section 8.

2 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 1, 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 special 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 the 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 back-end (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 computation are more efficient in the boolean representation, the EzPC compiler uses the Yao circuit builder ycirc to build the corresponding circuits (lines 19 to 24) (in our usage of ABY, boolean is synonymous with Yao, we elaborate in Section 6). We elaborate on cryptographic cost awareness in Section 3.

Using both arithmetic and boolean representations requires conversions between them. The EzPC compiler also instruments these conversion gates accordingly. For example, in line 17, the compiler converts \( a_{\text{acc}} \) to a boolean representation, before it is input to the comparison and multiplexer circuits.

3 Circuit generation and evaluation. The next step in EzPC 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.

2.1 Comparison with garbled circuits

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, 27, 34, 40, 44, 52]. Indeed, this is one of the main drivers behind the development of various specialized 2PC protocols in previous works [9, 40, 44] (we compare against them in Section 7). Here, we empirically demonstrate the performance benefits of mixed computations over boolean-only compute by comparing with state of the art Yao-based compilers CBMC-GC [29] and ObliVM [39].

The largest benchmark in CBMC-GC is a multiplication of two $8 \times 8$ matrices for which it generates about a million gates and takes about ten seconds to run. In contrast, EzPC uses arithmetic sharing, generates 1218 gates, and runs in less than 0.1 seconds. When we tried multiplying two $20 \times 20$ matrices with CBMC-GC, it timed out after 5 hours. Our benchmarks require much bigger computations (e.g., multiplying a $64 \times 576$ matrix with a $576 \times 1024$ matrix). Unlike CBMC-GC, ObliVM can scale to larger benchmarks (because it uses Yao-pipelining [30]). 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.

3 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 7).

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 & $\sim 20x$ more communication in a LAN setting). The size of a boolean multiplication circuit is quadratic in the bit-width, 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. Further implementation details can be found in Section 6.

4 FORMAL DEVELOPMENT

In this section we prove correctness and security of our EzPC compiler. The readers who are interested in using the compiler as a black box can move directly to Section 5 without loss of continuity. 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 model the 2PC back-end. 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 4.1). We combine this correctness theorem and the security of the 2PC back-end to prove security of the protocols generated by EzPC (Theorem 4.2). We present only selected parts of our formalization. Full definitions and auxiliary lemmas can be found in Appendix C. We will also release complete typeset proofs in a tech report.

![Figure 4: Source language syntax](image)

**Source language.** Our language is a simple imperative language shown in Figure 4. Types $\psi$ consist of the base types $\sigma$, and arrays of base types $\sigma[n]$, where $n$ is the array length. Though we model only one dimensional arrays, our implementation supports multi-dimensional arrays as well. 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 $e_1 > e_2$ (we support several other operators in the implementation, detailed in Section 6), conditionals $e ? e_1 : e_2$, array literals $[x][n]$, and array reads $x[e]$. The expression $1_{[n]}$ denotes input from party $j$. The statements $s$ in the language comprise

---

8We 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.
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.

![Figure 5: Source semantics](image)

### 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:

- **Value**: \( v : \Gamma \rightarrow c \)  
- **Runtime environment**: \( \rho : \Gamma \rightarrow v \)  
- **Observation**: \( O : \Gamma \rightarrow c \)

Values consist of constants and array of constants, runtime environment \( \rho \) maps variables to values, and observations are sequences of constants.

The judgment \( \rho ⊢ e \Downarrow v \) denotes the big-step evaluation of an expression \( e \) to a value \( v \) under the runtime environment \( \rho \). 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 \( [\alpha]_{n_1} \), and to a uint\( n \) value \( n \). It then returns \( c_n \), the \( n \)-th index value in the array, provided \( n < n_1 \), the length of the array. Rule (E-If) evaluates to some constant \( c \) denoting party \( j \)'s input. An array input can be written in the language as \( [\alpha]_{n_1} \), which can then evaluate using the rule (E-Array) (the notation \( \forall \in [n] \) is read as \( \forall \in \{0 \ldots n - 1\} \)). The remaining rules are straightforward, and are elided for space reasons.

The judgment \( \rho ⊢ x \Downarrow p_1 ; O \) represents the big-step evaluation of a statement \( s \) under environment \( \rho \), producing a new environment \( p_1 \) and observations \( O \). Rule (E-Decl) evaluates the expression \( e \) to \( v \), and returns the updated environment \( ρ, x \mapsto v \), with empty observations. The for statements evaluate through the internal while syntax. Specifically, the rule (E-For) appends \( p \) with \( x \mapsto n_1 \), evaluates while \( x \leq n_2 \) to \( p_1 ; O \), and returns \( p_1 - \{x\} \) (removing \( x \) from \( p_1 \) and \( O \)). Rule (E-Loop) shows the inductive case for while statements, when \( p(x) \leq n \). The rule evaluates \( s \), producing \( p_1 ; O_1 \). It then restricts \( p_1 \) to the domain of \( \rho (p_1) \) 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 \( \text{while} \), when \( p(x) > n \). Finally, the rule (E-Out) evaluates the expression, and adds its value to the observations.

### Secret label

\[ m ::= \mathcal{A} \mid \mathcal{B} \mid \mathcal{L} \]

\[ \ell ::= \mathcal{P} \mid m \]

\[ \sigma ::= \sigma^\ell [\sigma^m] \]

\[ e ::= c \mid [\alpha]_n \mid [\alpha]_{n_1} \mid [\alpha]_{n_2} \mid [\alpha]_{n_3} \mid [\alpha]_{n_4} \]

\[ \ell \mid \ell > m \]

\[ \ell := \ell \mid \ell > m \mid \ell \mid \ell \]

### Figure 6: Intermediate language syntax

#### 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 \( \ell \) 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 \( \tau \) are then labeled base types \( \sigma^\ell \) and arrays of labeled base types \( \sigma^\ell [\sigma^m] \). Most of the expression forms \( e \) are same as \( e \), except that the binary operators, and the conditional forms are annotated with labels \( \ell \). 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 \( \ell > m \) \( e \) denotes coercing \( e \) from label \( \ell \) 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 \( \ell \) are chosen non-deterministically. Section 6 describes the label inference scheme in our implementation.

The judgment \( \Gamma \vdash e : \tau \rightarrow \forall \alpha \), where \( \Gamma \) maps variables \( x \) to types \( \tau \), says that under \( \Gamma \), \( e \) (in the source language) compiles \( \forall \alpha \) (in the intermediate language), where \( \forall \alpha \) has type \( \tau \). Rules (T-Unlit) 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 \( \times \), or a secret arithmetic multiplication \( \times \). As our compiler is cryptographic cost aware, it never compiles the multiplication to boolean multiplication \( \times \). For conditional, \( \tau \) can be either public comparison, or secret boolean comparison \( \gg \). The rule for conditional (T-Cond) has two cases: when the conditional expression \( e \) is of type bool, both the branches have a base type \( \sigma^\ell \), for an arbitrary \( \ell \), and the conditional is compiled to a public conditional, whereas when the conditional expression has type bool \( \mathcal{B} \), it 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 \( \models e \in n \) to prove that the array index is in bounds. Rule (T-Inp) picks a label \( m \) for the input. Finally, the

---

1. Our language also has statements out \( i, e \) (resp., out \( i, e \) ) to reveal value of \( e \) to only the first (resp., second) party. We omit these for brevity.
2. Section 6 discusses our implementation of this check.
\[ \Gamma \vdash e : \tau \rightarrow \bar{e} \quad \Gamma \vdash x \rightarrow \bar{x} \mid \Gamma \]

**T-Uint**
\[ \Gamma \vdash n : \text{uint}^\ell \rightarrow n \]
\[ \Gamma \vdash e : \text{boolf}^\ell \rightarrow \bar{e} \]
\[ \Gamma \vdash e : \text{bool}^\ell \rightarrow e \]
\[ \Gamma \vdash e : \sigma^m \rightarrow \bar{e} \]

**T-Mult**
\[ \forall i \in [1, 2], \Gamma \vdash e_1 : \text{uint}^\ell \rightarrow \bar{e}_1 \]
\[ \Gamma \vdash e_2 : \text{uint}^\ell \rightarrow \bar{e}_2 \]
\[ \Gamma \vdash e_1 \cdot e_2 : \text{uint}^{\ell_1 + \ell_2} \rightarrow \bar{e}_1 \cdot \bar{e}_2 \]

**T-Read**
\[ \forall i \in [1, 2], \Gamma \vdash e_1 : \text{boolf}^\ell \rightarrow \bar{e}_1 \]
\[ \Gamma \vdash e \rightarrow \text{bool}^\ell \rightarrow \bar{e} \]
\[ \Gamma \vdash e : \text{boolf}^\ell \rightarrow \bar{e} \]
\[ \Gamma \vdash e : \sigma^m \rightarrow \bar{e} \]

**T-Cond**
\[ \Gamma \vdash x : \sigma^m \rightarrow \bar{x} \]
\[ \Gamma \vdash e : \text{boolf}^\ell \rightarrow \bar{e} \]
\[ \Gamma \vdash e_i : \sigma^m \rightarrow \bar{e}_i \]
\[ \ell = \sharp (\ell) \lor (\ell = \sharp (\bar{e})) \]
\[ \ell < \sharp (\ell) \lor (\ell = \sharp (\bar{e})) \]
\[ \ell > \sharp (\ell) \lor (\ell = \sharp (\bar{e})) \]

**T-Assign**
\[ \forall i \in [1, 2], \Gamma \vdash e_i : \sigma^m \rightarrow \bar{e}_i \]
\[ \ell = \sharp (\ell) \lor (\ell = \sharp (\bar{e})) \]
\[ \ell < \sharp (\ell) \lor (\ell = \sharp (\bar{e})) \]
\[ \ell > \sharp (\ell) \lor (\ell = \sharp (\bar{e})) \]

**T-Decl**
\[ \Gamma \vdash e : \psi^m \rightarrow \bar{e} \]
\[ \Gamma \vdash \psi \rightarrow \bar{\psi} \]
\[ \Gamma \vdash x \rightarrow \bar{x} \]
\[ \Gamma \vdash e : \sigma^m \rightarrow \bar{e} \]
\[ \Gamma \vdash e : \sigma^m \rightarrow \bar{e} \]

**Figure 7: Source compilation**

The subsumption rule (T-Sub) asserts that \( \sigma^m \) is an expression of \( \sigma^m \).

**Figure 8: Circuits syntax**

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 program and also flattens the arrays so that the circuits are unaware of the array structure. Crucially, the semantics do 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, input gates \( in^\ell \), multiplication gates \( \text{mult} \), comparison gates \( \text{gt} \), and multiplexer gates \( \text{mux} \), cohere gates \( \{l \rightarrow m\} \), and constants. Sub-circuits \( \bar{v} \) (generated from \( \bar{v} \)) then consist of gates and arrays of gates. A circuit \( x \) is either empty, binding of a circuit gate to \( w \) or \( \bar{w} \), or a sequence of circuits.

**Figure 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:

Circuit generation environment \( \bar{p} \) := \( \cdot | \bar{p}, x \mapsto \bar{c} \)

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

Statement evaluation \( \bar{p} \Downarrow \bar{c} \Downarrow \bar{v} \Downarrow \bar{e} \) evaluates a statement \( s \) under the environment \( \bar{e} \) to produce a new environment \( \bar{p}_1 \), and a circuit \( \bar{c} \). Rules (S-DECLC), (S-DECLCKT), and (S-DECLCKTA) show the variable declaration cases. Rule (S-DECLC) shows the case when \( \bar{e} \) evaluates to \( \bar{v} \), where \( \bar{v} \) is either a constant or an array of constants. In this case, the mapping \( x \mapsto \bar{v} \) is added to the environment, and the resulting circuit is empty. When \( \bar{e} \) evaluates to a sub-circuit \( g \), rule (S-DECLCKT) picks a fresh wire \( w \), adds the mapping \( x \mapsto w \) to the environment \( \bar{p} \), and outputs the circuit binding \( g \) \( w \). The rule (S-DECLCKTA) is analogous for \( \bar{e} \) 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 \( E_m \) and \( D_m \) (where \( m \) is either \( \mathcal{A} \) or \( \mathcal{B} \)). More concretely, \( E_m(c) \) returns a pair of strings \( (b_1, b_2) \) with the property that \( D_m(b_1, b_2) = c \). The string \( b_j \) denotes the \( j \text{th} \) party’s share of \( c \). We assume that the underlying 2PC protocol instantiates \( E_m \) and \( D_m \) appropriately. For ABY protocol [18], algorithms \( (E_m, D_m) \) (resp. \( (E_g, D_g) \)) correspond to the arithmetic (resp. boolean) secret-sharing and reconstruction algorithms.

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:

Circuit environment \( \bar{p} \) := \( \cdot | \bar{p}, w \mapsto b \)

Figure 10: Circuit semantics in a distributed runtime

We use \( \bar{p}_i \) to denote the circuit environment for party \( i \). We give the judgments \( \bar{p}_i \Downarrow_B g \Downarrow b_1, b_2 \) and \( \bar{p}_1 \Downarrow \chi \Downarrow \bar{p}_1', \bar{p}_2'; O \), where \( O \) are the observations (similar to source semantics). The former judgment evaluates a gate under the environments \( \bar{p}_1 \) and \( \bar{p}_2 \) generates shares \( b_1 \) of the gate’s output. Rule (C-In) evaluates the input gate \( \text{in}_m \), 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 \( \text{mult} \ g_1 \ g_2 \), the rule first evaluates \( \bar{g}_1 \Downarrow (b_1, b_2) \) and \( \bar{g}_2 \Downarrow (b_2, b_2) \). \( n_1 \) and similarly \( n_2 \) are combined to \( n_2 \). The final output of the mult gate is then \( E_m(n_1 \times n_2) \). Note that this is a functional description of how the mult gate evaluates, of course, \( n_1 \) and \( n_2 \) are not observed by the parties. Rule (C-COERC) creates the new shares using \( E_m \) (the corresponding rule for coercion from \( P \) 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-formed, and compiles to \( \bar{s} \) in the intermediate language, then \( s \) terminates in the source semantics with observations \( O \), \( \bar{s} \) evaluates to circuit \( \chi \), and \( \chi \) terminates in the circuit semantics with the same observations \( O \). Formally, the correctness theorem is as follows (the environments on the left of \( \Downarrow \) are empty and we elide the environments on the right of \( \Downarrow \) for brevity):

Theorem 4.1 (Correctness). \( \forall s, \bar{s}, \chi, \Downarrow s \Downarrow \bar{s} \Downarrow \bar{s} \Downarrow \chi \Downarrow \chi \Downarrow O \).

We prove the theorem in Appendix C.

Security theorem. The protocols we generate provide simulation-based security against a semi-honest adversary, in the framework of [12, 13, 23] 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 $Z$. An adversary $A$, selects and “corrupts” one of the parties - however, $A$ still follows the protocol specification. $A$ interacts with $Z$ that observes the view of the corrupted party. At the end of the interaction, $Z$ 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 $\Pi$ in the presence of $A$ and $Z$. Let $\text{REAL}_{A, Z, \Pi}$ denote the distribution ensemble describing $Z$’s output in this interaction. In the ideal interaction, parties send their inputs to a trusted functionality that performs the desired computation truthfully. Let $S$ (the simulator) denote the adversary in this ideal execution, and $\text{IDEAL}_{F, S, Z}$ the distribution ensemble describing $Z$’s output after interacting with the ideal adversary $S$. A protocol $\Pi$ is said to securely realize a functionality $F$ if for every adversary $A$ in the real interaction, there is an adversary $S$ in the ideal interaction, such that no environment $Z$, on any input, can tell the real interaction apart from the ideal interaction, except with negligible probability (in the security parameter $k$). More precisely, the above two distribution ensembles are computationally indistinguishable.

We shall assume a cryptographic 2PC backend that securely implements any circuit $\chi$ that is output by our compiler (see Figure 8). This means that for any well-typed source program $s$, let $\chi$ be the circuit generated as in Theorem 4.1. We assume that there exists a 2PC protocol $\Pi$ that securely realizes the functionality $\chi$ and let $S_{2pc}$ be the corresponding simulator (that runs on $\chi$, input of the corrupt party and the output obtained from trusted functionality for $\chi$). We note that ABY [18] provides such a protocol $\Pi$ and simulator $S_{2pc}$ for all circuits $\chi$ output in our framework. We now state and prove our security theorem.

**Theorem 4.2 (Security).** Let $s$ be a well typed program in our source language that generates a circuit $\chi$ (as defined in Theorem 4.1). Let protocol $\Pi$ be the two-party secure computation protocol that securely realizes $\chi$ (as defined above). Then, $\Pi$ securely realizes $s$.

**Proof.** Our simulator $S$ simply runs our compiler on program $s$ to obtain $\chi$. It is crucial that this compilation to circuits does not require the secret inputs of the parties. Next, $S$ 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 4.1, these outputs $O_1$ are identical to outputs (or observations) $O_2$ of $\chi$ under circuit semantics. Next, $S$ runs $S_{2pc}$ on $\chi$, input of the corrupt party and $O_2$. From the security of $\Pi$, we have that the simulated view output by $S_{2pc}$ is indistinguishable from the real view. Hence, the security follows.

## 5 SECURE CODE PARTITIONING

In this section, we describe our “secure code partitioning” technique that allows EzPC to execute programs that require large circuits. Our techniques take inspiration from the idea of pipelining Yao’s garbled circuits described in FastGC [30]. However, unlike FastGC, we do not operate at a circuit level and partitioning is independent of the specific 2PC protocol. Let $s$ be a program in our source language that generates a circuit $\chi$. For some programs, the circuit $\chi$ can be larger than the memory size $6$ and fail to execute. Partitioning enables us to execute such programs via a source to source transformation that is oblivious to the underlying 2PC backend. Partitioning decomposes the program $s$ into a sequence of smaller EzPC programs $t_1, t_2, \ldots, t_k$ (as defined 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 our partitioning scheme 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 $s$). More formal details follow, while an example illustrating the technique of secure code partitioning is provided in Appendix A.

Let $s$ be a program that 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 $q_0 = \bot$ (the public empty state). For all $1 \leq i \leq k$, $s_i$ 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 $\chi_1$ from $s_1$, the parties can execute $\chi_1, \chi_2, \ldots, \chi_k$ sequentially (in a distributed setting) to obtain $q_1, q_2, \ldots, q_k$, and finally output $z$. At the $i^{th}$ step, the parties only need to store information proportional to $x, y, q_{i-1}$ and $\chi_i$ (which is much smaller than $\chi$). 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 $t_1, t_2, \ldots, t_k$ as follows. Once again, define $q_0 = \bot$. Without loss of generality, let all $q_i$ be values in some additive ring $(\mathbb{Z}, +)$ (e.g., the additive ring $(\mathbb{Z}_{2^64}, +)$, the additive ring of integers modulo $2^{64}$). Let $r_1, \ldots, r_{k-1}$ 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 $1$. Alice’s output from $t_1$ is $r_1$. Next, every $t_i$ ($2 \leq i \leq k - 1$) takes as inputs $x, r_{i-1}$ from Alice and $y, q_{i-1}$ 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 $q_i + r_{i-1}$ 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 code partitioning:

**Theorem 5.1 (Correctness and Security of Partitioning).** 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 $\Pi_1, \Pi_2, \ldots, \Pi_k$.

---

6In 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.

7While 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.
such that for all \( i \), \( \Pi_i \) 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 \( \Pi_i \) 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 every \( 1 \leq i \leq k \), \( t_i \) is well-typed. By Theorem 4.2, \( \Pi \), 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 \( \Pi_i \) provides observations \( r_i \) to Alice and \( o_i \) to Bob. Protocol \( \Pi_k \) 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 \( z \).

**Implementing code partitioning.** We use partitioning for programs that require large circuits. Specifically, we first decompose the program \( s \) into a sequence of small programs \( s_1 \| \ldots \| s_k \). And then, \( \text{EzPC} \) generates sequence of programs \( t_1, \ldots, t_k \) automatically. We then compile and execute the 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 \( \text{EzPC} \) program. Resource analysis of high-level programs is a well-known hard problem [28] 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 \( \text{size}(u) \), we need to discuss three important cases: if \( u = u_1 \| u_2 \) then \( \text{size}(u) = \text{size}(u_1) + \text{size}(u_2) \); if \( u \equiv f(s_1, u_1, u_2) \) then \( \text{size}(u) = \max(\text{size}(u_1), \text{size}(u_2)) \); if \( s \equiv \text{for } i \in [n_1, n_2] \text{ do } u_1 \) then \( \text{size}(u) = (n_2 - n_1) \text{size}(u_1) \). If \( n_2 - n_1 \text{size}(u_1) \) is above the threshold then we replace \( u \) by

\[
\text{for } i \in [n_1, \left\lfloor \frac{n_1 + n_2}{2} \right\rfloor] \text{ do } u_1; \text{for } i \in [\left\lfloor \frac{n_1 + n_2}{2} \right\rfloor, n_2] \text{ do } u_1
\]

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

6 IMPLEMENTATION

We discuss some implementation details of \( \text{EzPC} \). The \( \text{EzPC} \) compiler is written in Python and compiles each of our benchmarks in under a second to C++ code that makes calls to the \( \text{ABY} \) library [18]. \( \text{ABY} \) provides support for Arithmetic computations based on [34], and boolean computations based on GMW [23] as well as Yao’s garbled circuits [58]. Although \( \text{EzPC} \) 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, \( \text{EzPC} \) 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. \( \text{ABY} \) provides multi-threading support (for the offline phase of the 2PC protocol); we leverage the support and use at most four threads in our evaluation.

\( \text{EzPC} \) 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 \( \text{EzPC} \). However, we have implemented integral division in 30 lines of \( \text{EzPC} \), while the floating-point support in \( \text{ABY} \) is under active development [17].

Some of our benchmarks require accessing arrays at secret indices. While \( \text{EzPC} \) enforces the array indices to be public, secret indices can be encoded in \( \text{EzPC} \) using multiplexers. For example, consider the expression \( A[x] \) where \( A \) is an array of size 2 and \( x \) is secret-shared. The developer can express this functionality in \( \text{EzPC} \) as \( x > 0 \Rightarrow A[1] : A[0] \). In general, a secret access to an array of size \( n \) requires \( n - 1 \) multiplexers in \( \text{EzPC} \).

We use an off-the-shelf solver (SeaHorn [26]) to check that the array indices are within bounds (\( i \leq e < n \)) in (T-READ) and (T-WRITE), Figure 7. We take the \( \text{EzPC} \) source program and translate it as an input C program to the solver. The solver takes less than a minute on our largest benchmark to verify that all the array accesses are in-bounds. This C program also enables validation of \( \text{EzPC} \) generated protocols via differential testing [42, 46].

Our implementation assigns the type labels (rule T-DECL) conservatively. Only the variables that govern the control flow, i.e., variables in if-conditions and for-loop counters are assigned public labels. All other variables are assigned arithmetic labels (that can later be coerced to boolean). We leave a more sophisticated type inference procedure for future work.

The compilation rules of Figure 7 can introduce repeated coercions from arithmetic to boolean and vice versa. Since \( \text{EzPC} \) 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 [2]. On each coercion, \( \text{EzPC} \) memorizes the pair of arithmetic share and boolean share involved in the coercion. \( \text{EzPC} \) invalidates such pairs when the variables corresponding to the shares are overwritten by assignments. In subsequent coercions, \( \text{EzPC} \) reuses valid pairs (if available) instead of inserting code to recompute them afresh. These optimizations are standard compiler optimizations [2], and we rely on their correctness (optimizations preserve outputs and well-typedness) to maintain the security of the optimized programs.

7 EVALUATION

We evaluate \( \text{EzPC} \) 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.

To begin, we first implement the benchmarks from Bost et al. [9] and MiniONN [40] (both of which study the same setting), and show that the performance of the high-level code written in \( \text{EzPC} \) is comparable to their hand-crafted protocols. Next, we demonstrate the generality and programmability aspects of \( \text{EzPC} \) by implementing state-of-the-art machine learning models from Tensorflow [1] and Bonsai [36]. Indeed, we provide the first 2PC implementation of Bonsai. We implement a Deep Neural Network (DNN) for CIFAR-10 dataset [35] from MiniONN [40] and matrix factorization [48] to evaluate partitioning.
Table 1: Linear classification results. We compare EzPC (LAN, WAN, Comm) with [9] (Prev time, Prev comm).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>d</th>
<th>Prev time (s)</th>
<th>Prev comm (KB)</th>
<th>LAN (s)</th>
<th>WAN (s)</th>
<th>Comm (KB)</th>
<th>Num gates</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breast cancer</td>
<td>50</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>25</td>
</tr>
<tr>
<td>Credit</td>
<td>47</td>
<td>0.3</td>
<td>0.4</td>
<td>0.3</td>
<td>0.3</td>
<td>36</td>
<td>795</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 2: Naïve Bayes results. We compare EzPC (LAN, WAN, Comm) with [9] (Prev time, Prev comm).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>n</th>
<th>Prev time (s)</th>
<th>Prev comm (MB)</th>
<th>LAN (s)</th>
<th>WAN (s)</th>
<th>Comm (MB)</th>
<th>Num gates</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nursery</td>
<td>5</td>
<td>1.5</td>
<td>0.2</td>
<td>0.4</td>
<td>0.6</td>
<td>73k</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Audiology</td>
<td>24</td>
<td>3.9</td>
<td>2.0</td>
<td>1.5</td>
<td>37</td>
<td>4219k</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Decision tree benchmarks. We compare EzPC (LAN, WAN, Comm) with [56] (Prev time, Prev comm).

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. Each machine has an Intel(R) Xeon(R) CPU E5-2673 v3 processor running at 2.40GHz with 28 GBs of RAM. When we compare our execution times with prior protocols, we match our system and network parameters with those of the prior work (as the code for works such as Bost et al. [9] and MiniONN [40] are not publicly available). Most of our benchmarks are related to machine learning and we set up (largely standard) notation and describe our benchmarks in Appendix B.

7.1 Secure prediction

Standard classifiers. We evaluate the three standard classifiers, linear, Naïve Bayes, and decision trees, from [9] on the following data sets from the UCI machine learning repository [38]: the Wisconsin Breast Cancer data set, Credit Approval data set, and ECG (electrocardiogram) classification data from [3].

The results for linear classification are in Table 1. 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. [9] 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 2. As before, n denotes the number of classes and F is the number of features. As before, we compare with Bost et al. [9] 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 3 compares against the more recent work of [56] on decision trees and further validates this claim.

Deep neural nets. We evaluate EzPC on the DNNs described in SecureML [44], Cryptonets [22], and the CNN from MiniONN [40]. For comparison, we consider their implementations from MiniONN [40], which outperforms their previous implementations. Table 4 shows the results8. 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 [44] (Figure 10 in [40]). It has three fully connected layers with square as the activation function. Next, we implement the DNN described in Cryptonets [22] (Figure 11 in [40]) 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 [40]), 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.

In Table 4, 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-shell" 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 partitioning and we discuss it in Section 7.2.

State-of-the-art classifiers. Tensorflow [1] is a standard machine learning toolkit. Its introductory tutorial describes two prediction models for handwritten digit recognition using the MNIST dataset [37]. 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

---

8MiniONN does not report the network round-trip time nor the bit-length of their inputs (we use 32-bit inputs).
The sixth stage performs a ReLU and a convolution. The final stage leaves for future work) and with this support, LOC would be lower and independent of depth.

We remark here that our current language does not support functions (which we leave for future work) and with this support, LOC would be lower and independent of depth.

7.2 Secure code partitioning

The largest benchmark of MiniONN [40] is a DNN for CIFAR-10 dataset [35]. 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 partitioning 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 5: Tensorflow tutorial benchmarks

<table>
<thead>
<tr>
<th>Classifier</th>
<th>LAN (s)</th>
<th>WAN (s)</th>
<th>Comm (MB)</th>
<th>Num And</th>
<th>Num Mul</th>
<th>Num gates</th>
<th>Model size</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regression</td>
<td>0.1</td>
<td>0.7</td>
<td>2</td>
<td>8k</td>
<td>35k</td>
<td>8k</td>
<td>85k</td>
<td>38</td>
</tr>
<tr>
<td>CNN</td>
<td>38.5</td>
<td>60.3</td>
<td>2955</td>
<td>4163k</td>
<td>42104k</td>
<td>3226k</td>
<td>122.6</td>
<td>172</td>
</tr>
</tbody>
</table>

Table 6: Bonsai benchmarks

<table>
<thead>
<tr>
<th>Dataset</th>
<th>LAN (s)</th>
<th>WAN (s)</th>
<th>Comm (MB)</th>
<th>Num And</th>
<th>Num Mul</th>
<th>Num gates</th>
<th>depth</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char5k</td>
<td>0.1</td>
<td>0.7</td>
<td>2</td>
<td>18k</td>
<td>3k</td>
<td>85k</td>
<td>1</td>
<td>89</td>
</tr>
<tr>
<td>USPS</td>
<td>0.2</td>
<td>0.9</td>
<td>4</td>
<td>62k</td>
<td>2k</td>
<td>25k</td>
<td>2</td>
<td>156</td>
</tr>
<tr>
<td>WARD</td>
<td>0.3</td>
<td>1.1</td>
<td>9</td>
<td>100k</td>
<td>8k</td>
<td>50k</td>
<td>3</td>
<td>283</td>
</tr>
</tbody>
</table>

Table 7: Partitioning results for matrix factorization. The time reported by [48] for this computation is 10440 seconds.

<table>
<thead>
<tr>
<th>Stage</th>
<th>LAN (s)</th>
<th>WAN (s)</th>
<th>Comm (MB)</th>
<th>depth</th>
<th>Num gates</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>175</td>
<td>662</td>
<td>29816</td>
<td>18370</td>
<td>32m</td>
<td>568</td>
</tr>
<tr>
<td>2</td>
<td>193</td>
<td>1095</td>
<td>31945</td>
<td>30916</td>
<td>37m</td>
<td>516</td>
</tr>
<tr>
<td>3</td>
<td>178</td>
<td>627</td>
<td>29810</td>
<td>16369</td>
<td>32m</td>
<td>478</td>
</tr>
<tr>
<td>Total</td>
<td>546</td>
<td>2384</td>
<td>91971</td>
<td>–</td>
<td>–</td>
<td>1494</td>
</tr>
</tbody>
</table>

Figure 11: Comparison of EzPC code with and without partitioning. x-axis denotes the number of layers in the DNN, while y-axis denotes time in seconds for the secure protocol.

Table 7 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 4, EzPC generated generic 2PC protocol is competitive with MiniONN here as well. Therefore, we believe that with partitioning, EzPC can scale to large 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 partitioned EzPC implementation would still succeed.

Scalability. To illustrate the scalability of partitioning, we evaluate a sequence of DNNs with and without partitioning in Figure 11. All layers are identical and partitioning places each layer in a separate stage. For DNNs with up to 4 layers, the performance, with and without partitioning, is almost identical and the lines overlap, thereby illustrating that partitioning does not cause any noticeable performance overheads. Memory issues start showing up in the non-partitioned implementation of the 5 layer DNN and it is slower. Performance degrades rapidly thereafter and DNNs with 6 or more layers fail to execute (terminate with a “bus error”). However, the partitioned implementation scales well to even these large DNNs.
7.3 Matrix factorization

EzPC is not tied to secure prediction and can express more general computations. To demonstrate this expressiveness, we implement secure matrix factorization [48]. Abstractly, given a sparse matrix \( M \) 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 \( M = UV \). This operator is useful in recommender systems. In particular, Nikolaenko et al. [48] 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 [48]). 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 \( M \) 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 [48] for one iteration is 2.9 hours. This computation is large enough that we partition each iteration into three stages. The first stage involves a Batch
t [4] 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 8. 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 [48] and in the WAN setting we are about 4 times faster. The main source of these significant speedups is that, unlike [48], 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 Batch
t’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 [48]) to scale a boolean circuits based backend to this benchmark.

7.4 Subsequent Work

Subsequent to our work, Juvekar et al. [33] have presented GAZELLE, a specialized protocol for DNNs. GAZELLE use a lattice-based packed additively homomorphic encryption scheme (PAHE) for arithmetic computations and garbled circuits for boolean computations. GAZELLE can evaluate the CNN benchmark of Table 4 in 0.8 seconds, as opposed to 5.1 seconds taken by EzPC with the ABY backend. Such advances in cryptographic backends are orthogonal to our contributions. In particular, once GAZELLE is available, we could add it as another cryptographic backend to EzPC. Furthermore, the authors of GAZELLE remark: “A final, very interesting and ambitious line of work would be to build a compiler that allows us to easily express arbitrary computations and automatically factor the computation into PAHE and two-party primitives” — the exact problem that EzPC solves.

8 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 (SDFL) [5, 41] and CBMC-GC [29] compile C or Pascal like programs into boolean circuits that are then evaluated using garbled circuits [58]. OhlviVM [39] protects access patterns using an oblivious RAM [24, 50] and also uses garbled circuits for compute. In Secure Multi-party Computation Language (SMCL) [47], Java like programs are compiled into arithmetic circuits that are then evaluated using the VIFF framework [15]. Wysteria [51] 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 [14, 23]. Mitchell et al. [43] allow the user to select between Shamir’s secret sharing [20] and fully homomorphic encryption [21]. 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 [27] allows mixing homomorphic encryption based arithmetic computations and garbled circuits based boolean computations and the interconver-
sions between the two are inserted by the programmer explicitly. The work of Kerschbaum et al. [34] provides a scheme to automatically assign homomorphic encryption or garbled circuits to each operator in a computation that is expressed as a sequence of dyadic operations. They conjecture that the problem is NP hard and gave a linear programming based solution and a quadratic time greedy heuristic. These techniques are not directly applicable to EzPC programs because of for-loops and if-conditions. How-
ever, we are exploring if these ideas can be extended to yield better type inference. Other examples include the VIFF framework [15] for arithmetic computations and Sharemind [8] (secure 3-party boolean computation).

2PC backends have made tremendous progress in the last decade. For example, the circuits can be optimized for depth [11, 17], large garbled circuits can be pipelined [30, 39], online complexity can be reduced at the cost of offline complexity [25], encrypted values output from a garbled circuit can be reused [45] and oblivious RAM [24, 50] can be used to hide access patterns of MIPS code [55]. 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, 6, 9, 10, 19, 31, 40, 44, 48, 49, 56].

9 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 generated protocols comprise combinations of arithmetic and boolean circuits and have performance compa-
rable to, or better than the previously known custom specialized
protocols from previous works. The compiler is backed by formal semantics that help it maintain correctness, security, and efficiency.

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. 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. Finally, we will explore the possibility of mechanically verifying the compiler implementation.

REFERENCES


B DESCRIPTION OF BENCHMARKS

We use $[N]$ to denote $\{0, 1, \ldots N-1\}$. Further, given a vector $x \in \mathbb{R}^d$, we say $\arg\max x = i$ if $x_i = \max \{x_0, \ldots, x_{d-1}\}$. Finally, if $A$ is a matrix (resp. vector) then we write $f(A)$ (resp. vector) obtained by applying the scalar function $f$ to each entry of $A$ pointwise.

We focus on the machine learning models for classification. A classifier $C$ uses a trained model to predict a label $\ell$ for an input data point $x$. For example, given a data point which is a tuple of humidity and temperature a classifier can predict a label "will rain" or "will not rain". The model size of a classifier is the number of parameters in the model. For example, the model size of the classifier in Figure 1 is $|w| + 1 = 31$. The accuracy of a classifier refers to the fraction of data points that the classifier labels correctly from a given set of test data points.

Standard classifiers. A binary linear classifier is one of the simplest classifiers. Here, the input is a data point $x \in \mathbb{R}^d$, and the model is a vector $w \in \mathbb{R}^d$. The possible labels are $\ell \in \{true, false\}$ and the classifier is $C_w \equiv w^T x > 0$. A more interesting classifier is Naïve Bayes [9] that predicts labels from the set $[n]$. Here, the input data point is a feature vector $x = (x_0, x_2, \ldots, x_{d-1})^T$ where each $x_j \in [F]$. The model size of this classifier is $\Theta(n d F)$.

A decision tree of size $N$ and depth $d$ takes as input an $x \in \mathbb{R}^d$ and the prediction task reduces to evaluation of a $d$-degree polynomial [9].

Deep neural nets. The next class of classifiers that we benchmark are deep neural nets or DNNs. A DNN has multiple layers such that each layer computes a matrix multiplication followed by an activation function $f$. The most common activation functions are square $f(x) = x^2$ and rectifier linear unit (ReLU) $f(x) = \max(x, 0)$. Given an input vector $x$, the predicted label of a DNN is

$$\operatorname{argmax} W_N \cdot f_{N-1}(\ldots f_1(W_1 \cdot x) \ldots)$$

Here, $f_i$’s are the (public) activation functions, the model consists of matrices $W_i$, $x \in \mathbb{R}^d$ is the input vector, and the operator $\cdot$ denotes a matrix multiplication. Neural nets usually have one or more fully connected layers, each of which multiplies a matrix with a vector. Some neural nets have convolution layers and such DNNs are also called Convolutional Neural Nets or CNNs. For the purpose of this paper, a convolution can be converted as a (heavy) matrix-matrix multiplication. The size of matrices manipulated by a convolution layer grows linearly with window size (typically 9 or 25), the number of output channels (typically 16, 32, or 64), and the size of the matrix input to this layer. Therefore, fully connected layers are lighter computation-wise compared to convolution layers. However, the model size of fully connected layers is larger than those of convolution layers. In general, DNNs are computationally heavy but provide much better accuracies on computer vision tasks than the classifiers discussed above.

State-of-the-art classifiers. Finally, there are a class of machine learning classifiers that are much more efficient than DNNs and provide reasonably good accuracies on standard learning tasks. Bonsai [36] is a state-of-the art classifier in this class and EzPC provides the first 2PC protocol for it. 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 $\theta_j$. 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

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

Here, $l_j(x)$ 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 sign($y$) otherwise.

In the following, we implement these classifiers in EzPC and report the time taken for making secure predictions. Ideally, the machine learning classifiers are mathematical expressions over $\mathbb{R}$ that are usually approximated by floating-point operations. As is standard, we port the classifiers to integer manipulating programs by scaling the models and rounding [40]. These ported classifiers are then implemented in EzPC.