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

Nishanth Chandran[1], Divya Gupta[1], Aseem Rastogi[1], Rahul Sharma[1], and Shardul Tripathi[2]

[1]Microsoft Research India
Email: {nichandr,t-digu,aseemr,rahsha}@microsoft.com

[2]Dept. of Computer Science and Engineering, I.I.T. Delhi, India
Email: 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* ([10], [50], [4], [38], [41]), 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 and x. Alice has following options.

She can program the computation in one of the several programmer friendly, domain-specific languages (such as Fairplay [39], Wysteria [46], ObliVM [37], CBMC-GC [28], SMCL [42], Sharemind [9], [40] 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 ([52], [24]) or as an arithmetic circuit ([21], [16], [22]). 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. 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. [19]) 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 inter-conversion gates between them. Clearly, writing correct and efficient protocols in such a framework is beyond an average programmer who does not understand the various trade-offs between arithmetic and boolean circuits, and even for an expert cryptographer, writing large computations in such a framework can be tedious and subject to errors (a sentiment echoed by Demmler et al. [19] themselves).

A third option for Alice is to earn a PhD in cryptography, and design and implement specialized, efficient 2PC protocols (similar to [10], [50], [38], [43]) for her tasks.

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

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

This paper presents EzPC[1], the first "cryptographic-cost aware" framework that generates efficient and scalable 2PC protocols from high-level programs devoid of any cryptographic details. The generated protocols use combinations of arithmetic and boolean circuits and have performance comparable to or better than the custom, specialized protocols from previous works [10], [50], [38], [41], [23], [43]. 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 the surprising fact that generic 2PC implementations are much more efficient than what they were believed to be. Below we describe the salient features of EzPC.

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 its 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.

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 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.

1. Read as "easy peasy", stands for Easy 2 Party Computation.

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[2]. 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 (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 4). 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 (or, "honest but curious") adversaries to the semi honest security of the 2PC back-end. We also prove a formal security theorem against semi-honest adversaries for our partitioning scheme (Section 3 and Section 4).

Evaluation. We have implemented EzPC using ABY [19] as the cryptographic back-end. 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 [1] and BONSAI [34], and also the matrix factorization example from Nikolaenko et al. [43]. 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 [34], 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 5 and Section 6).

Related Work. Before ABY [19], several works have proposed combining secure computation protocols based on

2. Using swap and disk space is feasible but it causes huge slowdown.

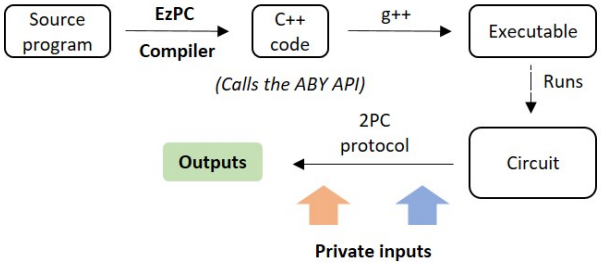

Figure 2: EzPC toolchain

homomorphic encryption and Yao's garbled circuits (e.g. [4], [7], [11], [20], [30], [43], [44], [47]), and some have also developed tools that allow writing such combinations (e.g. [8], [48], [27], [32]). However, as Demmler et al. [19] 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. We provide a detailed survey of related work in Section 7.

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 constitute 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 outputs the result of comparing `acc` with `b` 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, 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[3] for the example in Figure 1. 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

3. This is also how a program written directly in ABY would look like.

```
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);
   share *y_b = ycirc -> PutA2YGate(a_b);
19
   share *y_pred = ycirc -> PutGTGate(y_acc, y_b);
21 uint32_t one = 1;
   share *y_1 = ycirc -> PutCONSGate(one, bitlen);
23 uint32_t zero = 0;
   share *y_0 = ycirc -> PutCONSGate(zero, bitlen);
25 share *y_t = ycirc -> PutMUXGate(y_pred, y_1, y_0);

27 share *y_out = ycirc -> PutOUTGate(y_t, CLIENT);
   party -> ExecCircuit();
29
   if(role == CLIENT){ //only to the client
31   uint32_t_o = y_out -> get_clear_value<uint32_t>();
   }
```

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

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 20 to 25) (in our usage of ABY, boolean is synonymous with Yao, we elaborate in Section 5).

Using both arithmetic and boolean representations requires conversions between them. The EzPC compiler also instruments these conversion gates accordingly. For example, in lines 17 and 18, the compiler converts `a_acc` and `a_b` to boolean representation, before they are input to the comparison and multiplexer circuits.

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.

We can now concretely see the advantages of EzPC. Unarguably, it is easier for a developer to program and get right the code 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 once that's the case, it is more efficient to do both the addition and bitwise-or using boolean circuits (if the addition is still 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 she would have to revisit many parts of the circuit and change them.

To summarize, EzPC raises the level of abstraction for the programmer, and generates efficient 2PC protocols automatically, while its metatheory provides strong correctness and security guarantees.

3. Formal development

In this section we prove correctness and security of our EzPC 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 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 1). We combine this correctness theorem and the security of the 2PC back-end to prove security of the protocols generated by EzPC (Theorem 2). We present only selected parts of our formalization. Full definitions and proofs can be found in [3].

Source language. Our source 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. Though we model only one dimensional arrays, our implementation supports multi-dimensional

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

Figure 4: Source language syntax

$\boxed{\rho \vdash e \Downarrow v}$

$$\textsc{E-Var}\ \frac{}{\rho \vdash x \Downarrow \rho(x)} \qquad \textsc{E-Mult}\ \frac{\forall i \in \{1,2\}.\ \rho \vdash e_i \Downarrow n_i}{\rho \vdash e_1 \times e_2 \Downarrow n_1 \times n_2}$$

$$\textsc{E-Cond}\ \frac{\rho \vdash e \Downarrow c \quad c = \top \Rightarrow e_3 = e_1 \quad c = \bot \Rightarrow e_3 = e_2 \quad \rho \vdash e_3 \Downarrow c_3}{\rho \vdash e \,?\, e_1 \;:\; e_2 \Downarrow c_3}$$

$$\textsc{E-Read}\ \frac{\rho \vdash x \Downarrow [\overline{c_i}]_{n_1} \quad \rho \vdash e \Downarrow n \quad n < n_1}{\rho \vdash x[e] \Downarrow c_n} \qquad \textsc{E-Arr}\ \frac{\forall i \in [n].\ \rho \vdash e_i \Downarrow c_i}{\rho \vdash [\overline{e_i}]_n \Downarrow [\overline{c_i}]_n} \qquad \textsc{E-Inp}\ \frac{}{\rho \vdash \texttt{in}_j \Downarrow c}$$

$\boxed{\rho \vdash s \Downarrow \rho_1; O}$

$$\textsc{E-Decl}\ \frac{\rho \vdash e \Downarrow v}{\rho \vdash \psi\, x = e \Downarrow \rho, x \mapsto v; \cdot} \qquad \textsc{E-LoopT}\ \frac{\rho(x) > n}{\rho \vdash \texttt{while}\; x \leq n\; \texttt{do}\; s \Downarrow \rho; \cdot}$$

$$\textsc{E-LoopI}\ \frac{\rho(x) \leq n \quad \rho \vdash s \Downarrow \rho_1; O_1 \quad \rho_2 = [\rho_1]_{\mathsf{dom}(\rho)}[x \mapsto \rho_1(x) + 1] \quad \rho_2 \vdash \texttt{while}\; x \leq n\; \texttt{do}\; s \Downarrow \rho_3; O_2}{\rho \vdash \texttt{while}\; x \leq n\; \texttt{do}\; s \Downarrow \rho_3; O_1, O_2}$$

$$\textsc{E-If}\ \frac{\rho \vdash e \Downarrow c \quad c = \top \Rightarrow s = s_1 \quad c = \bot \Rightarrow s = s_2 \quad \rho \vdash s \Downarrow \rho_1; O}{\rho \vdash \texttt{if}(e, s_1, s_2) \Downarrow \rho_1; O}$$

$$\textsc{E-For}\ \frac{\rho, x \mapsto n_1 \vdash \texttt{while}\; x \leq n_2\; \texttt{do}\; s \Downarrow \rho_1; O}{\rho \vdash \texttt{for}\; x \;\texttt{in}\; [n_1, n_2]\; \texttt{do}\; s \Downarrow \rho_1 - \{x\}; O} \qquad \textsc{E-Out}\ \frac{\rho \vdash e \Downarrow c}{\rho \vdash \texttt{out}\; e \Downarrow \rho; c}$$

Figure 5: Source semantics (selected rules)

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 5), 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 and assignments ($\psi\, x = e$ and $x := e$ resp.), `for` loops, array writes ($x[e_1] := e_2$), `if` statements, and sequence of statements ($s_1; s_2$). 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

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.

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

Figure 6: Intermediate language syntax

programmer.

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 $n < n_1$, 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. Rule (E-IF) evaluates the guard expression, and then evaluates either s_1 or s_2 accordingly. The `for` statements evaluate through the internal `while` syntax. Specifically, the rule (E-FOR) appends ρ with $x \mapsto n_1$, evaluates `while` $x \leq n_2$ `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 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 5 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 5). 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 $\models e < n$ to prove that the array index is in bounds. Section 5 discusses our implementation of this check. 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 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. The rule (T-WRITE), similar to (T-READ), checks that the index has type $\mathtt{uint}^{\mathcal{P}}$, and is in bounds. Rule (T-OUT) types the

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

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

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

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

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

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

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

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

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

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

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

Figure 7: Compilation judgments (selected rules)

expression e at some secret label m. Rule (T-If) checks that the conditional expression is public, 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 $\texttt{uint}^{\mathcal{P}}$, 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

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

Figure 8: Circuits syntax

$\boxed{\widetilde{\rho} \vdash \widetilde{e} \Downarrow \widetilde{v}}$

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

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

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

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

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

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

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

$$\textsc{S-If}\ \frac{\widetilde{\rho} \vdash \widetilde{e} \Downarrow c \quad c = \top \Rightarrow \widetilde{s} = \widetilde{s_1} \quad c = \bot \Rightarrow \widetilde{s} = \widetilde{s_2} \quad \widetilde{\rho} \vdash \widetilde{s} \Downarrow \widetilde{\rho}_1; \chi}{\widetilde{\rho} \vdash \texttt{if}(\widetilde{e}, \widetilde{s_1}, \widetilde{s_2}) \Downarrow \widetilde{\rho}_1; \chi} \qquad \textsc{S-WriteCkt}\ \frac{\widetilde{\rho} \vdash x \Downarrow [\overline{w_i}]_n \quad \widetilde{\rho} \vdash \widetilde{e}_1 \Downarrow n_1 \quad n_1 < n \quad \mathsf{fresh}\ w \quad \widetilde{\rho} \vdash \widetilde{e}_2 \Downarrow g \quad \widetilde{\rho}_1 = \widetilde{\rho}[x \mapsto ([\overline{w_i}]_n[n_1 \mapsto w])]}{\widetilde{\rho} \vdash x[\widetilde{e_1}] := \widetilde{e}_2 \Downarrow \widetilde{\rho}_1; \texttt{bind}\ g\ w}$$

Figure 9: Evaluation of Intermediate Language to Circuit (selected rules)

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, 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 $\texttt{in}_j^m$, multiplication gates `mult`, comparison gates `gt`, and multiplexer `mux` gates, coerce gates $\langle \ell \triangleright m \rangle$, 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.

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 [19]. 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 ABY protocol [19], 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.

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.

Finally, 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 theorem states that 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. Therefore, well typed programs always terminate (array indices are always in bounds, there are no unbounded loops, etc.) and the 2PC protocol produces the same outputs as the source program.

Security theorem. The protocols we generate provide simulation-based security against a semi-honest adversary, in the framework of [24], [13], [14]. 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

$\boxed{\widehat{\rho}_1, \widehat{\rho}_2 \vdash g \Downarrow b_1, b_2}$

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

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

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

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

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

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

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

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

Figure 10: Circuit semantics in a distributed runtime (selected rules)

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 [19] 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.

4. Secure code partitioning

We now describe our "secure code partitioning" technique that allows EzPC to execute programs that require large circuits. 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[5] 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 details follow.

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 $1 \leq i \leq k-1$, 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 $q_0 =$

5. 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.

$\perp$. 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[6]. 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.

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. The following theorem states the correctness and security of the scheme.

Theorem 3 (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 $\Pi_1, \Pi_2, \ldots, \Pi_k$ 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. Then, 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 protocol Π_i provides observations r_i to Alice and o_i to Bob. And, 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.

Managing circuit sizes using partitioning. We use partitioning for programs that require large circuits. Specifically, the programmer decomposes the program s into a sequence of small programs $s_1||\ldots||s_k$ manually. And then, EzPC generates sequence of programs $t_1, \ldots t_k$ automatically. We then compile and execute the k programs $t_1, t_2, \ldots, t_k$ sequentially, freeing up memory usage after execution of each t_i. Automating the manual decomposition step requires an analysis that can statically estimate the resource usage of a EzPC program and we leave the construction of such an analysis as future work.

6. While the description of our protocol 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. To do so, we modify t_1 to output $o_1 = s_1 + r_1 + r'_1$ to both parties (where r'_1 is random and chosen by Bob). We can then appropriately modify the remaining steps as well.

5. Implementation

We discuss some implementation details of EzPC. The EzPC compiler is written in Python and compiles each of our benchmarks in under a second to C++ code that makes calls to the ABY library [19]. ABY provides a GMW-based [24] and a Yao-based [52] protocol for boolean computations. Although EzPC can generate code for both, we have observed better performance when using the latter and use it in our evaluation. We use the default ABY configuration for our experiment[7]. 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.

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 [18].

Some of our benchmarks require accessing arrays at secret indices. While EzPC enforces the array indices to be public, secret indices can be encoded in 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 EzPC as $x > 0\ ?\ A[1]\ :\ A[0]$. In general, a secret access to an array of size n requires $n-1$ multiplexers in EzPC.

We use an off-the-shelf solver (SeaHorn [26]) to bound check the array indices ($\models e < n$ in (T-Read) and (T-Write), Figure 7). We take the EzPC source program and translate it as an input to the solver. The solver 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 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 EzPC is aware of the cryptographic costs associated with these coercions, it tries to minimize them using several optimizations. For instance, if boolean shares of a variable are available in scope and the variable is involved in an addition then EzPC performs the addition in boolean rather than first coercing boolean to arithmetic and then performing an arithmetic addition. However, multiplication is always performed in arithmetic as the cost of a boolean multiplication is much higher than the cost of coercing from boolean

7. We set the security parameter to 128 and use OT extension-based arithmetic multiplication triples generation.

to arithmetic, performing an arithmetic multiplication, and then coercing from arithmetic to boolean [19]. Other optimizations include the standard "common subexpression elimination" optimization [2]. 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.

6. 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.

To begin, we first implement the benchmarks from Bost et al. [10] and MiniONN [38] (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 [1] and Bonsai [34]. Indeed, we provide the first 2PC implementation of Bonsai. We implement a Deep Neural Network (DNN) for CIFAR-10 dataset [33] from MiniONN [38] and matrix factorization [43] to evaluate partitioning.

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. Since most of our benchmarks are related to machine learning, we set up some notation (largely standard in machine learning) and describe our benchmarks next.

6.1. Benchmarks description

We use $[N]$ to denote $\{0, 1, \ldots, N-1\}$. Further, given a vector $x \in \mathbb{R}^d$, we say $\mathsf{argmax}\ x = i$ if $x_i = \mathsf{max}\ \{x_0, \ldots, x_{d-1}\}$. Finally, if A is a matrix (resp. vector) then we write $f(A)$ for the matrix (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 ℓ 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* [10] 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 $\Theta(ndF)$. 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 [10].

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) $f(x) = \mathsf{max}(x, 0)$. Given an input vector x, the predicted label of a DNN is

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

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 considered 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 [34] 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 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

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

Dataset	d	Prev. time (s)	Prev. Comm (kb)	LAN (s)	WAN (s)	Comm. (kb)	#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 1: Linear classification results. We compare EzPC (LAN, WAN, Comm) with [10] (Prev. Time, Prev. Comm).

Dataset	n	F	Prev. time (s)	Prev. Comm (kb)	LAN (s)	WAN (s)	Comm. (kb)	#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 2: Naïve Bayes results. We compare EzPC (LAN, WAN, Comm) with [10] (Prev. Time, Prev. Comm).

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 $\text{sign}(y)$ 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 [38]. These ported classifiers are then implemented in EzPC.

6.2. Secure prediction

Standard classifiers. We evaluate the three standard classifiers, linear, Naïve Bayes, and decision trees, from [10] on the following data sets from the UCI machine learning repository [36]: the Wisconsin Breast Cancer data set, Credit Approval data set, Audiology (Standardized) data set, Nursery data set, and ECG (electrocardiogram) classification data from [4].

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. [10] 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. [10] 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 [50] on decision trees and further validates this claim.

Dataset	d	N	Prev. time (s)	Prev. Comm (kb)	LAN (s)	WAN (s)	Comm. (kb)	#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 3: Decision tree benchmarks.We compare EzPC (LAN, WAN, Comm) with [50] (Prev. Time, Prev. Comm).

DNN	Prev. time (s)	Prev. Comm (kb)	LAN (s)	WAN (s)	Comm. (kb)	#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 4: DNN benchmarks. We compare EzPC (LAN, WAN, Comm) with [38] (Prev. Time, Prev. Comm).

Deep neural nets. We evaluate EzPC on the DNNs described in SecureML [41], Cryptonets [23], and the CNN from MiniONN [38]. For comparison, we consider their implementations from MiniONN [38], which outperforms their previous implementations. Table 4 shows the results[8]. 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 [41] (Figure 10 in [38]). It has three fully connected layers with square as the activation function. Next, we implement the DNN described in Cryptonets [23] (Figure 11 in [38]) 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 [38]), 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

8. MiniONN does not report the network round-trip time nor the bit-length of their inputs (we use 32-bit inputs).

Classifier	LAN (s)	WAN (s)	Comm. (Mb)	#And	#Mul	#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 5: Tensorflow tutorial benchmarks

Dataset	LAN (s)	WAN (s)	Comm. (Mb)	#And	#Mul	#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 6: Bonsai benchmarks

protocols, for both the LAN and the WAN settings. Further, lines of EzPC source code required is still small. We note that MINIONN and EzPC implementations are both based on ABY. These results are surprising as MINIONN has a much more efficient preprocessing phase and has SIMD (single instruction multiple data) capabilities that allow it to perform matrix operations efficiently. In contrast, EzPC focuses on generic 2PC, and such optimizations are part of our future work. 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 6.3.

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 [35]. 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 5.

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 partitioning.

We next present BONSAI [34] results on three standard datasets: character recognition (Chars4k [17], accuracy 74.71%), text recognition (USPS [31], accuracy 94.4%), and object categorization (WARD [51], accuracy 95.7%). The BONSAI models are already over integers and no porting from floating-point is required. We implement the trained classifiers in EzPC for all the benchmarks from [34], and show the representative results in Table 6. Out of all the benchmarks from [34], the dataset WARD requires the largest model. The column "depth" shows the depth of the tree used by BONSAI. The size of EzPC program grows with the depth of the tree, as the straightforward EzPC implementation requires a loop for each layer of the tree[9].

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)	#And	#Mul	#Gates
Total	265.6	647.5	40683	21m	61m	337m
Stage 6	55.2	122.6	6744	12m	10m	98m

TABLE 7: Partitioning results for CIFAR-10. MINIONN takes 544 seconds and communicates 9272 Mb.

Stage	LAN (s)	WAN (s)	Comm. (Mb)	depth	#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 8: Partitioning results for matrix factorization. The time reported by [43] for this computation is 10440 seconds.

6.3. Secure code partitioning

The largest benchmark of MINIONN [38] is a DNN for CIFAR-10 dataset [33]. 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 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 arbitrary sized 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.

6.4. Matrix factorization.

EzPC is not tied to secure prediction and can express

9. 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.

more general computations. To demonstrate this expressiveness, we implement secure matrix factorization [43]. 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. [43] 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 [43]). 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 [43] for one iteration is 2.9 hours[10]. This computation is large enough that we partition each iteration into three stages. The first stage involves a Batcher [5] 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 [43] and in the WAN setting we are about 4 times faster. The main source of these significant speedups is that, unlike [43], 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 miniscule compared to the mammoth implementation effort put in by Nikolaenko et al. (Section 5 of [43]) to scale a boolean circuits based backend to this benchmark.

7. 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) [39], [6] and CBMC-GC [28] compile C or Pascal like programs into boolean circuits that are then evaluated using garbled circuits [52]. ObliVM [37] protects access patterns using an oblivious RAM [45], [25] and also uses garbled circuits for compute. In Secure Multiparty Computation Language (SMCL) [42], Java like programs are compiled into arithmetic circuits that are then evaluated using the VIFF framework [16]. Wysteria [46] 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 [15], [24]. Mitchell et al. [40] allow the user to select between Shamir's secret sharing [21] and fully homomorphic encryption [22]. 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 [19] falls in this category. The TASTY tool [27] 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. The work of Kerschbaum et al. [32] 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 as well as a quadratic time greedy heuristic. These techniques are not directly applicable to EzPC programs because of `for`-loops and `if`-conditions. However, we are exploring whether these ideas can be extended to yield a better type inference. Other examples include the VIFF framework [16] for arithmetic computations and Sharemind [9] (that provides secure 3-party boolean computation).

The implementations of 2PC backends have made tremendous progress in the last decade. For example, the circuits can be optimized for depth [18], [12], large garbled circuits can be pipelined [29], and oblivious RAM [45], [25] can be used to hide access patterns of MIPS code [49]. Incorporating these backends would only improve the performance and scalability of EzPC implementations.

Many works have designed specialized, hand-crafted protocols for various 2PC tasks. This requires deep knowledge of cryptography to ensure the security of the protocols. Examples include privacy-preserving classification of medical ElectroCardioGram (ECG) [4], iris and fingerprint recognition [7], remote diagnostics [11], private sequence analysis [20], private biometric identification [30], privacy-preserving matrix factorization and ridge-regression [43], [44], machine-learning classification [10], decision trees and forests [50], neural network training [41] and prediction [38].

8. Conclusion

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 comparable to, or better than the custom, specialized protocols from previous works. The 2PC backed we use provides security against semi-honest or passive adversaries. Given a 2PC backend secure against malicious or active adversaries, Theorem 2 can be extended

10. [43] does not report the network round-trip time.

to obtain a framework with malicious security. Furthermore, partitioning can be modified to provide malicious security by *signing* the shares of the intermediate states. However, we are not aware of a maliciously secure 2PC implementation for combinations of arithmetic and boolean circuits.

References

[1] 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*, abs/1603.04467, 2016. 2, 10, 12

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. 10

[3] A. Anonymous. Full proofs of EzPC, 2017. 4

[4] M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A. Sadeghi, and T. Schneider. Secure evaluation of private linear branching programs with medical applications. In *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, pages 424–439, 2009. 1, 3, 11, 13

[5] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, 1968. 13

[6] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 257–266, 2008. 13

[7] M. Blanton and P. Gasti. Secure and efficient protocols for iris and fingerprint identification. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, pages 190–209, 2011. 3, 13

[8] 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*, PETShop '13, pages 23–26, 2013. 3

[9] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, pages 192–206, 2008. 1, 13

[10] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser. Machine learning classification over encrypted data. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015. 1, 2, 10, 11, 13

[11] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel. Privacy-preserving remote diagnostics. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 498–507, 2007. 3, 13

[12] N. Büscher, A. Holzer, A. Weber, and S. Katzenbeisser. Compiling low depth circuits for practical secure computation. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part II*, pages 80–98, 2016. 13

[13] R. Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, 2000. 7

[14] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145, 2001. 7

[15] 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 *Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings*, pages 416–432, 2012. 13

[16] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Public Key Cryptography - PKC 2009, 12th International Conference on Practice and Theory in Public Key Cryptography, Irvine, CA, USA, March 18-20, 2009. Proceedings*, pages 160–179, 2009. 1, 13

[17] T. E. de Campos, B. R. Babu, and M. Varma. Character recognition in natural images. In *VISAPP 2009 - Proceedings of the Fourth International Conference on Computer Vision Theory and Applications, Lisboa, Portugal, February 5-8, 2009 - Volume 2*, pages 273–280, 2009. 12

[18] D. Demmler, G. Dessouky, F. Koushanfar, A. Sadeghi, T. Schneider, and S. Zeitouni. Automated synthesis of optimized circuits for secure computation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1504–1517, 2015. 9, 13

[19] D. Demmler, T. Schneider, and M. Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015. 1, 2, 3, 7, 8, 9, 10, 13

[20] 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.*, 17(3):117–128, Feb. 2013. 3, 13

[21] R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and fact-track multiparty computations with applications to threshold cryptography. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 101–111, 1998. 1, 13

[22] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178, 2009. 1, 13

[23] 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 *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 201–210, 2016. 2, 11

[24] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229, 1987. 1, 7, 9, 13

[25] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996. 13

[26] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The seahorn verification framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 343–361, 2015. 9

[27] W. Henecka, S. Kögl, A. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 451–462, 2010. 3, 13

[28] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ANSI C. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 772–783, 2012. 1, 13

[29] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 35–35, Berkeley, CA, USA, 2011. USENIX Association. 13

[30] Y. Huang, L. Malka, D. Evans, and J. Katz. Efficient privacy-preserving biometric identification. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011. 3, 13

[31] J. J. Hull. A database for handwritten text recognition research. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(5):550–554, 1994. 12

[32] F. Kerschbaum, T. Schneider, and A. Schröpfer. Automatic protocol selection in secure two-party computations. In *Applied Cryptography and Network Security - 12th International Conference, ACNS 2014, Lausanne, Switzerland, June 10-13, 2014. Proceedings*, pages 566–584, 2014. 3, 13

[33] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009. 10, 12

[34] A. Kumar, S. Goyal, and M. Varma. Resource-efficient machine learning in 2 KB RAM for the internet of things. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 1935–1944, 2017. 2, 10, 12

[35] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010. 12

[36] M. Lichman. UCI machine learning repository, 2013. 11

[37] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 359–376, 2015. 1, 13

[38] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 24th ACM Conference on Computer and Communications Security, CCS 2017, Dallas, Texas, USA, October 30 - Nov 3, 2017*, 2017. 1, 2, 10, 11, 12, 13

[39] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 287–302, 2004. 1, 13

[40] J. C. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman. Information-flow control for programming on encrypted data. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 45–60, 2012. 1, 13

[41] P. Mohassel and Y. Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38, 2017. 1, 2, 11, 13

[42] J. D. Nielsen and M. I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007, San Diego, California, USA, June 14, 2007*, pages 21–30, 2007. 1, 13

[43] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. Privacy-preserving matrix factorization. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 801–812, 2013. 1, 2, 3, 10, 12, 13

[44] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 334–348, 2013. 3, 13

[45] R. Ostrovsky. Efficient computation on oblivious rams. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 514–523, 1990. 13

[46] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 655–670, 2014. 1, 13

[47] A. Schröpfer and F. Kerschbaum. Forecasting run-times of secure two-party computation. In *Eighth International Conference on Quantitative Evaluation of Systems, QEST 2011, Aachen, Germany, 5-8 September, 2011*, pages 181–190, 2011. 3

[48] A. Schröpfer, F. Kerschbaum, and G. Müller. L1 - an intermediate language for mixed-protocol secure computation. In *Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference, COMPSAC 2011, Munich, Germany, 18-22 July 2011*, pages 298–307, 2011. 3

[49] X. S. Wang, S. D. Gordon, A. McIntosh, and J. Katz. Secure computation of MIPS machine code. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part II*, pages 99–117, 2016. 13

[50] D. J. Wu, T. Feng, M. Naehrig, and K. E. Lauter. Privately evaluating decision trees and random forests. *PoPETs*, 2016(4):335–355, 2016. 1, 2, 11, 13

[51] J. Yang, Y. Li, Y. Tian, L. Duan, and W. Gao. Group-sensitive multiple kernel learning for object categorization. In *IEEE 12th International Conference on Computer Vision, ICCV 2009, Kyoto, Japan, September 27 - October 4, 2009*, pages 436–443, 2009. 12

[52] A. C. Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167, 1986. 1, 9, 13