PicoZK: Integrated, Efficient, and Extensible Zero-Knowledge Proof Engineering

Joseph P. Near
jnear@uvm.edu
University of Vermont

Onyinye Dibia
Onyinye.Dibia@uvm.edu
University of Vermont

David Darais
darais@galois.com
Galois, Inc.

Mark Blunk
mark@stealthsoftwareinc.com
Stealth Software Technologies, Inc.

1 INTRODUCTION

PicoZK is a framework for building zero-knowledge proof (ZKP) statements. PicoZK enables the programmer to specify a statement using high-level constructs, and it compiles the statement into a lower-level representation that forms the input to a zero-knowledge proof systems. PicoZK has the following design goals:

- **Integrated**: PicoZK embedded in a popular general-purpose programming language (Python), with smooth integration between the host language and the statement.
- **Efficient**: PicoZK is able to take advantage of recent advances in ZKP backends (e.g. RAM) to improve performance.
- **Accessible**: PicoZK is designed to be usable by programmers with limited ZKP background.
- **Extensible**: PicoZK provides programmers with tools to build higher-level abstractions.

This document is an introduction to the design, implementation, and use of of PicoZK. It describes how to use the framework to build statements, the built-in abstractions PicoZK provides, and how programmers can use PicoZK to build new abstractions. In addition, the document describes several applications of PicoZK that we have built during its development. These applications provide a roadmap for how to use PicoZK to build practical ZKP statements.

2 OVERVIEW

PicoZK is implemented as a Python library. The library provides a compiler that generates ZKP statements, and a number of datatypes that can be used by the programmer to specify statements. The following table provides an overview of the datatypes described in this document, including references to the sections where they are covered.

Type	Section
Built-in Wire Types	
`Wire`	3
`ArithmeticWire`	3
`BooleanWire`	4
`BinaryWire`	5
Built-in Abstract Datatypes	
Binary Integers (`BinaryInt`)	5
Read/Write Arrays (`ZKRAM`)	7
Derived Datatypes	
Lists	3
Matrices & Tensors	6
Oblivious Lists	7
Oblivious Stacks	7

Each of these datatypes builds on previous datatypes to provide higher levels of abstraction to the programmer. In many cases, applications will require programmers to build their own custom datatypes to provide specific ZKP-friendly functionality. PicoZK is designed for this kind of extensionality, and many of the built-in datatypes are implemented using PicoZK itself. Figure 1 summarizes the low-level datatypes that PicoZK provides, and also lists some of the higher-level abstract datatypes built on top of them.

Each section of this document introduces a new feature of a PicoZK abstraction, and each section includes a case study which demonstrates the feature's use to build a ZKP statement. The following table provides an overview of these case studies, including references to the sections where they are covered.

Application	Section
Hashing & commitments	3
Data analytics with Pandas	4
SHA256 Hashing for Binary Data	5
Neural networks with PyTorch	6
Stable matching with Gale-Shapley	7

The source code for PicoZK can be found on GitHub.[1] The system requires Python 3.x and also interoperates with several popular Python libraries. It produces ZKP statements in the SIEVE IR0 intermediate representation format, which can be used as input to several different ZKP systems.

3 ARITHMETIC CIRCUITS

PicoZK implements an eager compiler for ZK statements by defining new abstract datatypes whose operations emit (as a side effect) gates in an arithmetic circuits. The lowest level of abstraction in PicoZK is the `Wire` object, which provides exactly the same operations as the underlying arithmetic circuit: addition and multiplication. The (simplified) definition appears in Figure 2, along with `SecretInt`, a utility for adding a constant to the statement's witness.

Datatype: `Wire`
Represents a wire in a circuit over an arbitrary finite field. Stores wire name, value, and field order.
Operations: `+`, `*`

Simple circuits. With just this definition, it is possible to write Python programs that compile to arithmetic circuits. For example, the following code adds the number 5 to the witness, then constructs an arithmetic circuit with one addition and one multiplication gate:

[1] https://github.com/uvm-plaid/picozk

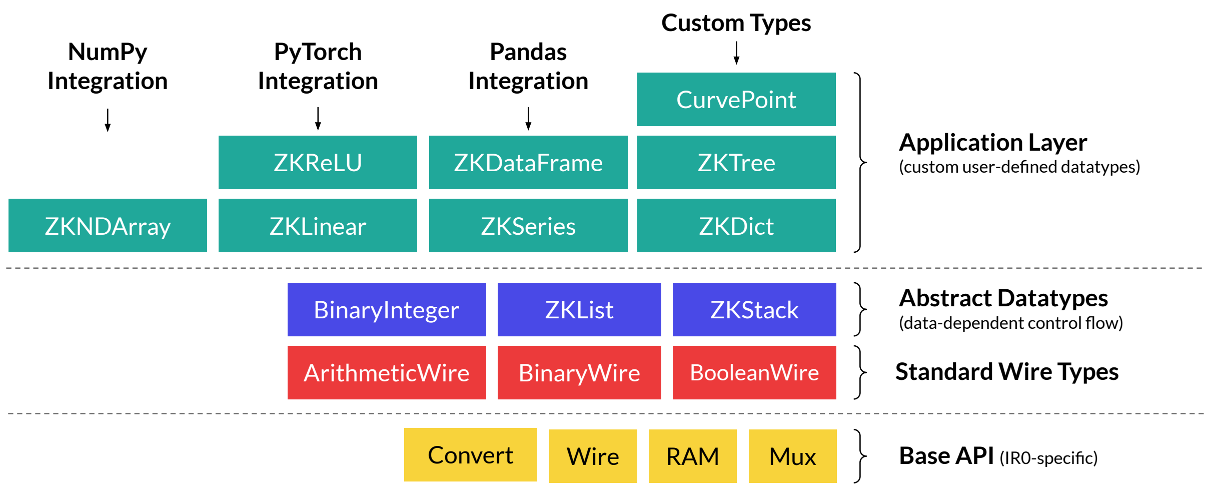

Figure 1: Summary of PicoZK's datatypes. The bottom layer provides datatypes corresponding to the low-level output of the compiler. PicoZK provides abstract datatypes built on this bottom layer to enable high-level specification of ZKP statements. The top layer contains custom datatypes that are implemented using PicoZK itself, and PicoZK is easy to extend with new custom datatypes.

```
@dataclass
class Wire:
  wire: str
  val: int
  field: int

  def __add__(self, other):
    if isinstance(other, Wire):
      r = cc.emit_gate('add', self.wire, other.wire)
    elif isinstance(other, int):
      r = cc.emit_gate('addc', self.wire, f'< {other % self.field} >')

    return Wire(r, self.val + val_of(other) % self.field, self.field)

  def __mul__(self, other):
    if isinstance(other, Wire):
      r = cc.emit_gate('mul', self.wire, other.wire)
    elif isinstance(other, int):
      r = cc.emit_gate('mulc', self.wire, f'< {other % self.field} >')

    return Wire(r, self.val * val_of(other) % self.field, self.field)

def SecretInt(x):
  assert x % cc.field == x
  return cc.add_to_witness(x % cc.field)
```

Figure 2: Core PicoZK: `Wire` objects represents wires in an arithmetic circuit; operations on `Wire` objects eagerly compile to gates in the ZK statement. The `SecretInt` function constructs a `Wire` from a constant by adding the constant to the witness and returning a `Wire` object.

```
x = SecretInt(5)
z = x + x * x
print(z)
```

```
Wire(wire='$2', val=30)
```

The output is a `Wire` object that stores both the resulting value (30) and the name of the wire corresponding to that result in the constructed circuit. In the SIEVE-IR representation, the resulting circuit is:

```
$0 <- @private();
$1 <- @mul($0, $0);
$2 <- @add($0, $1);
```

Building proof statements. The definition above allows constructing arithmetic circuits with inputs provided in a witness file. To construct complete ZK statements, PicoZK provides a compiler interface that constructs complete SIEVE-IR relation and witness files, plus utilities for revealing a wire's value to the verifier. The `assert0` function reveals to the verifier that its input wire has the value 0, and is defined as follows:

```
def assert0(x):
  cc.emit_gate('assert_zero', x.wire, effect=True)
```

We can combine the use of `assert0` with the compiler interface to construct a complete ZK statement from the earlier example:

```
with PicoZKCompiler('equals_30', field=2**61-1):
  x = SecretInt(5)
  z = x + x * x
  assert0(z + -30)
```

The compiler assumes a finite field of order `field`; the default is $2^{61} - 1$. Running this program results in three files:

(1) `equals_30.rel`, the *relation*, which is known to both prover and verifier, and defines the arithmetic circuit's structure.
(2) `equals_30.type0.wit`, the *witness*, which is known to only the prover, and which defines the secret input values to the circuit.
(3) `equals_30.type0.ins`, the *instance*, which is known to both prover and verifier, and which defines the public inputs to the circuit. PicoZK does not use the instance, so it will always be empty.

Building on the abstraction. Next, we build on the `Wire` abstraction to add functionality, by implementing new operations *in Python*, leveraging the abstract operations provided by the existing datatype. In particular, we can add support for negation, subtraction, and exponentiation (by a constant) by writing code that *appears to be standard Python*. We define a new datatype, `ArithmeticWire`, to support these operations.

Datatype: `ArithmeticWire`

Represents a wire in a circuit over a large finite field. Stores wire name, value, and field order.

Operations: `+`, `*`, `-`, `**` (exponentiation)

```
class ArithmeticWire(Wire):
  def __neg__(self):
    return self * (self.field - 1)

  def __sub__(self, other):
    return self + (-other)
  __rsub__ = __sub__

  def __pow__(self, n):
    def exp_by_squaring(x, n):
      if n%2 == 0:
          if n // 2 == 1:
              return x * x
          else:
              return exp_by_squaring(x * x,  n // 2)
      else:
          return x * exp_by_squaring(x * x, (n - 1) // 2)

     assert isinstance(n, int)
     return exp_by_squaring(self, n)
```

We implement exponentiation by squaring, to optimize the number of gates in the resulting circuit. The implementation is via a natural recursive function, exactly like the one we might use for concrete values. When run with an `ArithmeticWire` as input, this function produces an arithmetic circuit that implements the procedure.

Case study: Poseidon hash function. A common pattern in ZK statements is *commit-and-prove*: the prover commits to a particular input, and then incorporates validation of the commitment into the statement itself. A common way to accomplish the commit-and-prove pattern is to implement a ZK-friendly hash function in the statement, hash the input values, and reveal the hash to the verifier.

We implement the ZK-friendly Poseidon hash function [2] by *adapting the (non-ZK) reference implementation*[2] from the original Python. Because of our abstractions, this process requires only minimal changes to the original code.

The implementation appears in Figure 3. We omit the setup functions for the (public) hash parameters (e.g. `mds_matrix` and `rc_field`), which are copied *unchanged* from the reference implementation. Because these are public parameters, the existing Python code can be run by the prover to generate the parameters during the statement generation process. This kind of smooth integration between unmodified Python code and PicoZK code is a major advantage of our approach.

To use the hash function, we call the `hash` function on a vector of `Wire` objects. For example, the following PicoZK program reads

[2] https://github.com/ingonyama-zk/poseidon-hash

```
class PoseidonHash:
  ...
  def s_box(self, element):
    return pow(element, self.alpha)

  def full_rounds(self):
    for r in range(0, self.half_full_round):
      for i in range(0, self.t):
        self.state[i] = self.state[i] + self.rc_field[self.rc_counter]
        self.rc_counter += 1
        self.state[i] = self.s_box(self.state[i])
      self.state = np.dot(self.state, self.mds_matrix)

  def partial_rounds(self):
    for r in range(0, self.partial_round):
      for i in range(0, self.t):
        self.state[i] = self.state[i] + self.rc_field[self.rc_counter]
        self.rc_counter += 1
      self.state[0] = self.s_box(self.state[0])
      self.state = np.dot(self.state, self.mds_matrix)

  def hash(self, input_vec):
      self.rc_counter = 0
      padded_input = input_vec + [0 for _ in range(self.t-len(input_vec))]
      self.state = [padded_input[i] + self.state[i] for i in range(self.t)]
      self.full_rounds()
      self.partial_rounds()
      self.full_rounds()
      return self.state[1]
```

Figure 3: PicoZK implementation of Poseidon hash function [2], with minimal adaptation from reference implementation.

in a text file from disk, converts each character of the file to a secret integer in the witness, and hashes the vector of characters (in blocks of size `t`).

```
with open('secret_data_file.txt', 'r') as f:
  data = f.read()

with PicoZKCompiler('poseidon_hash'):
  poseidon_hash = PoseidonHash(p, alpha = 17, input_rate = t, t = t)
  secret_data = [SecretInt(ord(c)) for c in data]
  blocks = [secret_data[i * t:(i + 1) * t] for i in \
            range((len(secret_data) + t - 1) // t )]
  for block in blocks:
      digest = poseidon_hash.hash(block)
  assert0(digest - digest.val)
```

The resulting statement will encode an arithmetic circuit that hashes the witness and reveals the digest to the verifier. This approach works because the arithmetic operations defined in the methods of the `PoseidonHash` class will result in new gates added to the circuit, based on the definitions of the corresponding operators in the `Wire` class. This case study also demonstrates smooth integration with several important Python features:

- **Lists**: the secret (witness) data is represented as a list of `Wire` objects, and standard Python list operations (e.g. indexing, list comprehensions) can be used.
- **Loops**: standard Python loop constructs (e.g. `for`) can be used, as long as the loop parameters are public (i.e. not `Wire` objects). The resulting circuit encodes the unrolling of the loop.
- **Libraries**: the implementation represents public parameters as NumPy matrices, and uses NumPy operations (e.g. `np.dot`). NumPy matrices are designed to contain arbitrary types, including `Wire` objects, and PicoZK integrates smoothly with many operations implemented by NumPy (and other libraries).

4 BOOLEANS AND COMPARISONS

Many applications require equality comparisons, which are not possible using arithmetic circuit operations alone. It is possible to perform these comparisons by adding additional values to the witness or by converting arithmetic-valued wires into binary representation (explored in Section 5). For simple equality comparisons, however, the SIEVE-IR includes a plugin (the `mux` plugin) that allows ZK backends to optimize the implementation of equality checks. This section describes how we can expose this plugin in PicoZK, to provide maximum efficiency for equality checks in arithmetic circuits.

Utilizing the `mux` plugin. The SIEVE-IR includes a plugin that provides gates of the following form:

```
$3 <- @call(mux, $0, $1, $2);
```

If the wire `$0` has the value 0, then the `mux` call sets the wire `$3` to `$1`; otherwise, it sets `$3` to `$2`. We can use this construct to test equality between two values by placing the difference between them in the input wire (`$0`). We implement this approach in PicoZK by adding the following definition of equality to the `ArithmeticWire` class:

```
def __eq__(self, other):
  diff = self - other
  r = cc.emit_gate('call', 'mux', diff.wire,
                                  cc.wire_of(1),
                                  cc.wire_of(0))
  return Wire(r, int(val_of(self) == val_of(other)), self.field)
__req__ = __eq__
```

This implementation sets the output wire to the value 1 if the equality test is true, and 0 if it is false. The `wire_of` function is a helper that converts constants into (public) wire values in the circuit, with caching to re-use existing wires when possible.

Boolean wires & operators. Our equality test always returns 0 or 1, but as an element of the same large field used by the rest of the circuit. We represent values like these using the `BooleanWire` datatype, which also defines standard boolean operators designed to work for these values.

Datatype: `BooleanWire`

Represents a wire with a value of 0 or 1, in a circuit over a large finite field. Stores wire name, value, and field order.

Operations: `+`, `*`, `&`, `|`, `~`, `to_arith`

These operations can be easily defined using operations on the underlying wires:

```
class BooleanWire(Wire):
  def __and__(self, other):
    return self * other
  def __or__(self, other):
    return (self * other) + (self * (~other)) + ((~self) * other)
  def __invert__(self):
    return (self * (self.field - 1)) + 1
  def to_arith(self):
    return ArithmeticWire(self.wire, self.val, self.field)
```

Since boolean values are actually embedded in the same large finite field as the rest of the circuit, conversion from a boolean wire into an arithmetic one does not add to the circuit. However, we require the programmer to perform this conversion manually to avoid accidental use of boolean operators on values other than 0 or 1.

```
1  from picowizpl import *
2  from picowizpl.poseidon_hash import PoseidonHash
3  import pandas as pd
4
5  p = 2**61-1
6  with PicoWizPLCompiler('miniwizpl_test', field=p):
7    df = pd.read_csv('ma2019.csv')
8
9    # Convert the PUMA code to an integer
10   # PUMA 25-00703 is transformed to the number 1
11   #   Essex County East (Salem, Beverly, Gloucester, Newburyport)
12   df['PUMA'], codes = df['PUMA'].factorize()
13
14   # OWN_RENT column is 1 if individual owns their home, 2 if they rent
15   # add the data to the witness
16   sdf = df[['PUMA', 'OWN_RENT']].applymap(SecretInt)
17
18   # Hash the data and reveal the digest
19   poseidon_hash = PoseidonHash(p, alpha = 17, input_rate = 3)
20   digest = poseidon_hash.hash(list(sdf['PUMA']) + list(sdf['OWN_RENT']))
21   reveal(digest)
22
23   # Filter in the rows that have the correct PUMA and OWN_RENT values
24   owners = sdf.apply(lambda r: (r['PUMA']==1) & (r['OWN_RENT']==1),
25                      axis=1).sum()
26   renters = sdf.apply(lambda r: (r['PUMA']==1) & (r['OWN_RENT']==2),
27                       axis=1).sum()
28   reveal(owners)        # reveal statistic
29   reveal(renters)       # reveal statistic
```

Figure 4: Proof of a statistical computation using the Pandas library. The program loads a dataset containing an excerpt of US Census data from Massachusetts, hashes the data and reveals the digest, and calculates the total number of home owners and renters in a specific Census area (PUMA).

Finally, we define a `mux` function for arithmetic circuits that expects its first argument to be a boolean value:

```
def mux(a, b, c):
  return a.to_arith() * b + (~a).to_arith() * c
```

Case study: statements about statistics with Pandas. Figure 4 contains a complete PicoZK program that uses the Pandas library to calculate summary statistics of a dataset containing US Census data.[3] The program calculates the total number of homeowners and renters in a particular Census area (a PUMA), and reveals both results to the verifier.

The Pandas library is designed for statistical computations over tabular data stored in *dataframes*. The library allows dataframes to contain arbitrary objects—including PicoZK `Wire` objects. After preprocessing the data, the program converts each data element into a `Wire` (line 16). The program hashes the data (lines 19-21), then calculates the desired statistics and reveals them (lines 24-29).

Many of the built-in Pandas operations work correctly on dataframes that contain `Wire` objects because they invoke operations of the underlying objects. For example, the `sum` method on lines 25 and 27 calls the addition method on `Wire` objects, resulting in addition gates in the circuit.

However, not all Pandas operations can be used unmodified. Pandas programs typically filter rows by using a form of indexing, but this approach requires each comparison to return a Python `bool` (not a `Wire`). As a result, our program performs the filtering

[3]https://media.githubusercontent.com/media/usnistgov/SDNist/main/nist%20diverse%20communities%20data%20excerpts/massachusetts/ma2019.csv

manually (using the apply method on lines 24 and 26). Section 6 demonstrates how methods in existing libraries can be overridden to allow the idiomatic approach to work with PicoZK.

The dataset used in this example has 7,635 rows; the PicoZK compiler takes about 10 seconds to produce the circuit, which has about 9.7 million multiplication gates (of which about 2 million are for hashing).

5 BINARY CIRCUITS AND FIELD SWITCHING

The approach for comparisons and boolean operations described in Section 4 works only for equality—it cannot support inequality comparisons (greater-than, less-than, etc). PicoZK can support inequality comparisons via *field switching*—converting an element in a large field into a binary representation in the binary field $GF(2)$. The SIEVE-IR provides a field switching gate, which ZK backends can use to provide efficient implementations of field switching. ZK backends that support this feature typically do not provide support for switching between arbitrary fields; instead, they support only the binary field ($GF(2)$) and a large field used in the arithmetic circuit (e.g. $GF(2^{61}-1)$). Similarly, PicoZK provides support only for a single large field and the binary field.

Converting to and from binary representations. To provide this support, we add a method called to_binary to the ArithmeticWire class. The method constructs a gate like the following in the output circuit (for a default field of size $2^{61}-1$):

```
1: $2 ... $62 <- @convert(0: $0);
```

This gate converts a single arithmetic field element (wire $0) to an array of binary values (wires $2 through $62). The number of wires needed for the binary representation depends on the size of the arithmetic field.

To work with binary representations, PicoZK provides the BinaryWire and BinaryInt classes. The BinaryWire class represents a wire holding a single bit, while the BinaryInt class stores a list of binary wires. The BinaryInt class defines the is_negative method, which returns a single bit indicating whether the underlying integer is negative (i.e. it returns the most significant bit).

Datatype: BinaryWire
Represents a wire with a value of 0 or 1, in $GF(2)$. Stores wire name, value, and field order.
Operations: +, *, to_bool

Datatype: BinaryInt
A binary representation of an integer, as a list of BinaryWire objects.
Operations: ==, <<, >>, is_negative, to_arith

```
@dataclass
class BinaryInt:
  wires: List[BinaryWire]

  def is_negative(self):
    return self.wires[0]
```

To convert back to an arithmetic field element, PicoZK provides the to_bool method on BinaryWire objects. Invoking this method on a binary wire results in a conversion gate to an arithmetic field element whose value is 0 or 1, represented as a boolean in PicoZK. PicoZK also provides the to_arith method on BinaryInt objects, which converts an integer represented in binary into an arithmetic field element.

Defining binary circuits. BinaryWire objects allow the definition of binary circuits in the ZK statement, in the same way as ArithmeticWire objects enable the construction of arithmetic circuits. For example, we can add the following method to the BinaryInt class to allow equality comparisons between binary representations of integers:

```
def __eq__(self, other):
  ok = 1
  for w1, w2 in zip(self.wires, self._wires_of(other)):
    ok = ok * ((w1 * w2) + ((w1 + 1) * (w2 + 1)))
  return ok
```

This method defines a binary circuit over the wires representing the underlying integer. Its output will be 1 if the two integers are equal, and 0 otherwise. A similar approach can be used to construct standard circuits that operate on binary-represented integers (e.g. adders, multipliers, shifts, etc).

Implementing binary operators. Our primary motivation for converting arithmetic field elements into a binary representation is to support inequality checks. The most efficient way to perform these checks is to compute the difference between the two inputs (in an arithmetic circuit) and convert the difference to its binary representation to check if it is negative. We add the following methods to the ArithmeticWire class:

```
class ArithmeticWire(Wire):
  ...
  def is_negative(self):
    return self.to_binary().is_negative().to_bool()
  def __lt__(self, other):
    return (self - other).is_negative()
  def __gt__(self, other):
    return (other - self).is_negative()
```

The is_negative method converts the arithmetic field element to its binary representation, calls the is_negative method on that representation to find its most significant bit, and then converts that bit into a boolean wire. The greater-than and less-than operators are easy to define in terms of this operation.

We can also define commonly-used circuits on binary representations of integers. For example, a simple adder circuit can be added to the BinaryInt class; shifting and rotation can be implemented by simply adjusting wire positions within the representation—no new gates are required in the circuit.

```
class BinaryInt:
  ...
  def __add__(self, other):
    out_wires = []
    carry = 0

    for a, b in zip(reversed(self.wires),
                    reversed(self._wires_of(other))):
      ab = a + b
      out = ab + carry
      out_wires.append(out)
      carry = (a * b) + (ab * carry)
    return BinaryInt(list(reversed(out_wires)))

  def __rshift__(self, n):
    bw = len(self.wires)
    return BinaryInt([0]*n + self.wires[:bw-n])
```

```
class ZKSHA256:
  def __init__(self):
    self._h = [BinaryInt(util.encode_int(x, 2**32)) for x in _h]

  def maj(self, x, y, z):
    return (x & y) ^ (x & z) ^ (y & z)

  def ch(self, x, y, z):
    return (x & y) ^ ((~x) & z)

  def compress(self, chunk):
    w = [BinaryInt([0 for _ in range(32)]) for _ in range(64)]
    w[0:15] = chunk

    for i in range(16, 64):
      s0 = w[i-15].rotr(7) ^ w[i-15].rotr(18) ^ (w[i-15] >> 3)
      s1 = w[i-2].rotr(17) ^ w[i-2].rotr(19) ^ (w[i-2] >> 10)
      w[i] = (w[i-16] + s0 + w[i-7] + s1)

    a, b, c, d, e, f, g, h = self._h
    k = [BinaryInt(util.encode_int(x, 2**32)) for x in _k]

    for i in range(64):
      s0 = a.rotr(2) ^ a.rotr(13) ^ a.rotr(22)
      t2 = s0 + self.maj(a, b, c)
      s1 = e.rotr(6) ^ e.rotr(11) ^ e.rotr(25)
      t1 = h + s1 + self.ch(e, f, g) + k[i] + w[i]

      h = g
      g = f
      f = e
      e = (d + t1)
      d = c
      c = b
      b = a
      a = (t1 + t2)

    for i, (x, y) in enumerate(zip(self._h, [a, b, c, d, e, f, g, h])):
      self._h[i] = (x + y)

  def hash(self, msg):
    padding_amount = 512 - ((len(msg) + 1 + 64) % 512)
    padded_msg = msg + [1] + [0]*padding_amount + \
                 util.encode_int(len(msg), 2**64)
    assert len(padded_msg) % 512 == 0
    chunks = [padded_msg[i:i+512] for i in range(0, len(padded_msg), 512)]
    chunk_words = [[BinaryInt(chunk[i:i+32])
                    for i in range(0, len(chunk), 32)]
                   for chunk in chunks]
    for chunk in chunk_words:
      self.compress(chunk)
    return self._h
```

Figure 5: SHA256 hash function, implemented in PicoZK. We omit the definitions of the constants H and K.

```
  def rotr(self, n):
    bw = len(self.wires)
    return BinaryInt(self.wires[bw-n:] + self.wires[:bw-n])
```

Case study: SHA256 hash. SHA256 is a commonly-used hash function for binary data. When the input to a PicoZK statement is already represented in a binary format, it can be useful to commit to the input via a SHA256 hash (e.g. because its SHA256 hash may be used as part of a public-key signature). We implement the SHA256 hash function in PicoZK by adapting an existing Python implementation. We use `BinaryInt` objects to represent the 32-bit words used in SHA256. The full implementation appears in Figure 5.

6 ABSTRACT DATATYPE: TENSORS

Our next application is a ZK statement that a (secret) trained machine learning model produces a particular output when performing inference on a specific input. This ability is useful in the context of machine-learning-as-a-service (MLaaS), where a service provider may wish to prove to the client that they indeed performed inference correctly, without revealing the model itself. The service provider may also wish to prove to a regulator that their trained model satisfies a fairness or accuracy criterion on a public benchmark dataset.

Challenges to smooth integration. Deep learning libraries like PyTorch use *tensors*—or *n*-dimensional arrays—as the basic building blocks for deep learning models. An ideal solution for producing ZK statements about these models would be to simply replace the tensors representing the model's parameters (or weights) with tensors containing `ArithmeticWire` objects, and then run inference as normal to obtain `ArithmeticWire` objects representing the prediction.

However, due to their focus on performance, deep learning libraries (including PyTorch) typically do not allow tensors to contain arbitrary datatypes; instead, they are allowed to contain only the datatypes supported by the underlying high-performance CPU and GPU backends used by the library.

As a result, we will need to use our own representation for tensors, as a NumPy *n*-dimensional array of `ArithmeticWire` objects. Then, we will extend existing operations in the deep learning library to work for our representation.

Encoding floating-point numbers. An additional challenge of this application is the use of floating-point numbers in model parameters. We adopt a fixed-point encoding for these numbers as arithmetic field elements, whose precision is controlled by a scale parameter. Here, we use the field size $2^{61} - 1$. We also define functions for encoding *n*-dimensional tensors using NumPy arrays.

```
SCALE = 1000
p = 2**61-1
def encode(x):
    return int(SCALE*x) % p

encode_tensor = np.vectorize(encode)
encode_zk_tensor = np.vectorize(lambda x: SecretInt(encode(x)),
                                otypes=[ArithmeticWire])
```

Case study: multi-layer perceptron inference with PyTorch. Our simplest model is a multi-layer perceptron, which can be implemented using tensor multiplications alone. For this model, only one layer type is required: PyTorch's `nn.Linear` layer. A linear layer computes $xA^T + b$ for input x, layer weights A, and layer bias b.

For our ZK statement, we would like the layer's weights and bias to be secret values known only to the prover. However, as described earlier, PyTorch's `Tensor` datatype cannot contain `ArithmeticWire` objects. To encode the ZK statement, we define a new layer type—a `ZKLinear` layer—which acts like PyTorch's linear layer during training, but encodes its weights as a NumPy array of `ArithmeticWire` objects during inference, and uses NumPy's matrix operations to compute its results.

```
class ZKLinear(torch.nn.Linear):
  def forward(self, inp):
    if not self.training and not hasattr(self, 'zk_weights'):
        self.zk_weight = encode_zk_matrix(self.weight.detach().numpy())
        self.zk_bias = encode_zk_matrix(self.bias.detach().numpy())

    return inp @ self.zk_weight.T + self.zk_bias * self.bias_scale
```

Here, the `forward` method is overridden to use NumPy operations; on the first inference, the layer encodes its weights and bias using the approach described above. The bias term is scaled to ensure that its scale matches the result of the matrix multiplication. We can use the new layer type to define our model, setting the bias scale parameter for each layer appropriately.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.ZKLinear(784, 32)
        self.fc2 = nn.ZKLinear(32, 10)
        self.fc1.bias_scale = SCALE
        self.fc2.bias_scale = SCALE*SCALE

    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        return x
```

To perform inference, we instantiate the model and load the model weights from disk (assuming we have pre-trained the model). Then, we encode the inputs (`test_inputs`) and run the model.

```
with PicoZKCompiler('pytorch_mlp', field=p):
  model = Net()
  model.load_state_dict(torch.load('mnist.pt'))
  model.eval()

  encoded_input = encode_matrix(test_inputs.detach().numpy())
  output = model(encoded_input)

  for output_probs in output:
    for x in output_probs:
      reveal(x)
```

For 10 inputs, the PicoZK compiler takes about 4.5 seconds to produce the circuit, which has about 250,000 multiplication gates. For 100 inputs, the compiler takes about 20 seconds, and the circuit has about 2.5 million multiplication gates.

Case study: deep neural network inference with PyTorch. Our previous approach is limited to neural networks without non-linear activation functions (like ReLU) because these functions cannot be expressed as arithmetic circuits. To add support for ReLU, we can implement it as a binary circuit, and then leverage field switching as described in Section 5. We implement a `ZKReLU` layer:

```
class ZKReLU(torch.nn.ReLU):
  def forward(self, inp):
    is_pos = np.vectorize(lambda x: (~x.is_negative()).to_arith(),
                          otypes=[ArithmeticWire])
    return is_pos(inp) * inp
```

The PicoZK compiler takes about the same amount of time to generate the circuit as in the previous example; the number of multiplication gates increases slightly, to about 255,000 for 100 inputs.

7 ABSTRACT DATATYPE: RAM

Programs and algorithms typically assume that random-access memory (RAM) is available, so ZK statements about programs and algorithms often need to encode properties of RAMs (e.g. in the form of arrays). We saw examples earlier of PicoZK programs that include Python lists—but we only used publicly-known values as indices into those lists. When the statement includes RAM-like structures with indexing by secret values (e.g. encoded as `ArithmeticWire` objects), then it cannot be encoded into an arithmetic circuit in a straightforward way. Consider the following example:

```
idx = SecretInt(3)
values = [1,2,3,4,5]
result = values[idx]
```

The naive solution is to construct an arithmetic circuit that performs a linear scan of the RAM-like structure, and outputs the value of the element whose index matches the desired one. Unfortunately, this approach results in extremely large circuits—for a RAM-like structure of size n, it requires at least $O(n)$ comparisons *per RAM access.*

More sophisticated approaches can reduce this overhead significantly. To take advantage of this capability, the SIEVE IR provides a RAM plugin similar to the mux plugin described in Section 4. PicoZK exposes the functionality of this plugin via the `RAM` abstract datatype.

The `ZKRAM` datatype. PicoZK provides RAM support via an abstract datatype that supports `read` and `write` operations on a fixed-size RAM. Values stored in RAM are hidden from the verifier, and indices used for reading and writing can be (public) Python integers or inlineArithmeticWire objects.

Datatype: `ZKRAM`

A read/write memory of (public) fixed length, with indexing by secret values.

Operations: `read`, `write`, `len`

Building on the abstraction: oblivious arrays. We can use the `ZKRAM` datatype to build oblivious data structures that are similar to standard Python structures, but that support indexing by secret values. For example, PicoZK provides an oblivious array implemented as follows:

```
class ZKList:
    def __init__(self, xs):
        self.ram = ZKRAM(len(xs))
        for i, x in enumerate(xs):
            self.ram.write(i, SecretInt(x))

    def __getitem__(self, idx):
        return self.ram.read(idx)

    def __setitem__(self, idx, val):
        self.ram.write(idx, val)

    def __len__(self):
        return len(self.ram)
```

The implementation simply delegates to an underlying `ZKRAM` object for reads and writes. We can use this implementation to support the earlier example:

```
idx = SecretInt(3)
values = ZKList([1,2,3,4,5])
result = values[idx]
```

Building on the abstraction: oblivious stacks. We can also use the `ZKRAM` abstraction to build other kinds of oblivious data structures, including stacks. A stack can be implemented using a `ZKRAM` to hold the elements, plus an index representing the current

```
def index(mat, val, start, length):
  result = -1
  for offset in range(length):
    result = mux(mat[start + offset] == val, start + offset, result)
  return result

def prefers(w, m, m_p):
    v_m   = index(preference_matrix, m,   w*NUM_PREFS, NUM_PREFS)
    v_m_p = index(preference_matrix, m_p, w*NUM_PREFS, NUM_PREFS)
    return v_m < v_m_p

def gale_shapley(preference_matrix):
  unmarried_men = ZKStack(len(men))
  for x in men:
    unmarried_men.push(SecretInt(x))

  marriages = ZKList([NO_MARRIAGE for _ in women])
  next_proposal = ZKList([0 for _ in men])

  for i in range(ITERS):
    m = unmarried_men.pop()
    w = preference_matrix[m*NUM_PREFS + next_proposal[m]]
    next_proposal[m] += 1

    wi = w-len(men)

    # branch 1
    b1 = marriages[wi] == NO_MARRIAGE
    v = mux(b1, m, marriages[wi])
    marriages[wi] = v

    # branch 2
    b2 = prefers(w, m, marriages[wi])
    b22 = b2 & (~ b1)
    unmarried_men.cond_push(b22, marriages[wi])
    marriages[wi] = mux(b22, m, marriages[wi])

    # branch 3 (else)
    b3 = (~ b1) & (~ b2)
    unmarried_men.cond_push(b3, m)

  return marriages
```

Figure 6: Gale-Shapley in PicoZK.

top of the stack. Our oblivious stack can provide conditional push and pop operations, by using muxes to conditionally adjust both the top of the stack and its elements.

```
class ZKStack:
  def __init__(self, max_size):
    self.ram = ZKRAM(max_size)
    self.top = 0

  def cond_pop(self, cond):
    self.top = mux(cond & ~(self.top == 0), self.top - 1, self.top)
    return mux(cond, self.ram.read(self.top), 0)

  def cond_push(self, cond, v):
    self.top = mux(cond, self.top + 1, self.top)
    self.ram.write(self.top, mux(cond, v, self.ram.read(self.top)))
```

Many other kinds of oblivious data structures can be constructed in a similar way, by building on the `ZKRAM` abstraction.

Case study: stable matching with Gale-Shapley. The Gale-Shapley algorithm [1] solves the *stable matching problem* (or *stable marriage problem*): given two groups of n individuals each, and a *preference list* for each individual, produce a bijection between the groups such that there is no pair of individuals which prefers each other (according to the preference list) to their partners according to the bijection. Such algorithms have been used to match medical residents to hospitals (in the National Resident Matching Program) and to match organ donors with potential recipients.

In the context of sensitive preference lists, it may be desirable to prove that a particular bijection is actually a stable match, without revealing any individual's preferences. Proving in ZK that a particular bijection is an output of the Gale-Shapley algorithm (and thus a stable match) is challenging because the algorithm makes repeated use of data-dependent control flow in the form of stack and list operations. In PicoZK, we can adapt a simple Python implementation of the algorithm using the abstract data types described in this section.

The complete implementation appears in Figure 6. It uses `ZKList` objects to represent the preference lists (as a dense matrix), the current set of marriages, and the next proposal each individual will make. The implementation uses a `ZKStack` object to hold the remaining unmarried men; at the start of the algorithm, it contains all of the men. The data-dependent control flow of the algorithm itself (expressed in Python using `if` statements) is translated into uses of `mux`.

For 500 individuals, preference lists of length 15, and 1000 iterations of the algorithm, PicoZK takes about 5 seconds to compile the statement. The statement contains about 8.3 million multiplication gates, and 86,000 RAM operations.

8 CONCLUSION

This paper describes PicoZK, a framework for building zero-knowledge proof (ZKP) statements. PicoZK provides high-level constructs for constructing these statements, and compiles statements into lower-level circuit representations. In addition, PicoZK is specifically designed to be extensible with new kinds of abstract datatypes that enable new applications. We have released PicoZK as open-source.[4]

ACKNOWLEDGMENTS

The authors would like to thank Kimberlee Model, Sam Dittmer, Mark Blunk, Will Yang, Seoyeon Hwang, Steve Lu, Emma Shroyer, Mayank Varia, and Harunobu Ishii for their contributions to the development of PicoZK. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited). This material is based upon work supported by DARPA under Contract No. HR001120C0087 and by the Broad Agency Announcement Program and the Cold Regions Research and Engineering Laboratory (ERDC-CRREL) under Contract W913E521C0003. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or ERDC-CRREL.

REFERENCES

[1] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.
[2] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for {Zero-Knowledge} proof systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 519–535, 2021.

[4]https://github.com/uvm-plaid/picozk