The VampIR Book

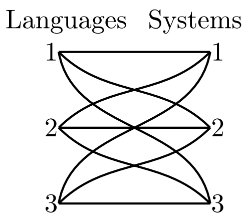
The VampIR Team

Contents

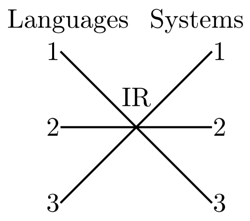
1 What is VampIR and How is it Used?

1.1 Why VampIR?

VampIR is, at its core, a language for defining arithmetic circuits over finite fields. It's intended to be a universal intermediate language supporting many proof systems based on arithmetic circuits. Any higher-level language which intends to compile to arithmetic circuits may target VampIR as an intermediate language.



The need for an intermediate language is obvious. Without an adequate intermediate, systems must support every desired proof system individually. This creates an ecosystem like that depicted in the right figure, where the total work required for interconnectedness is quadratic. This also creates more points of failure and more opportunities for support inconsistencies.



With an adequate intermediate representation, the total amount of work necessary to fill out the ecosystem is only linear. This can be seen in the ecosystem depicted in the left figure. Languages can, instead of targeting a specific proof system, target the intermediate representation. That language would then have support for every proof system targeted by that intermediate representation.

VampIR's goal is to fulfill this need by providing a minimal and expressive interface for describing the core data structure used by most modern zero-knowledge-proof systems. To that end, VampIR aims to be flexible and easy to use but doesn't provide any cryptographic features of its own. It does not presuppose any particular implementation or design for a proof system. VampIR files are sufficiently generic that they may even be used for applications that use arithmetic circuits but are not cryptographic in nature.

1.2 Using VampIR

A very basic example of a VampIR program is the following;

```
def x = 10;

x = 10;
```

Here, we can see the two main top-level commands available in VampIR. That first line defines a constant, 'x', to which we assign the value '10'. The second line is an equation that is expected to be true. Every arithmetic circuit is interpreted as a proposition. Specifically, it is the proposition corresponding to the truth of all the equations appearing in the file which generated it. In this case, the compiled circuit will merely check that '$10 = 10$'. Notice that every line must end in a semicolon. VampIR does not generally care about white space or newlines (beyond spaces separating individual tokens); this example would be interpreted the same if all newlines were removed and everything was put on a single line.

To compile this circuit we, of course, need VampIR up and running. We first clone VampIR's directory.

```
$ git clone git@github.com:anoma/vamp-ir
```

VampIR is implemented in Rust and can easily be compiled from source using Rust Cargo.

```
$ cd vamp-ir
$ cargo build
```

This will create the VampIR binary at '/target/debug/vamp-ir' within the main VampIR directory.

VampIR does not possess any cryptographic capabilities on its own. This means that some specific parameters, such as the field size, cannot be determined by VampIR, but are instead decided at compile time. To compile this circuit, we first must choose a target. For this example, I will choose PLONK [What is the name of the program? How does one access other targets?].

For the sake of this example, I will assume that the lines in our example program are saved into a file called 'ex1.pir', stored within a new folder called 'examples' within the main VampIR directory.

```
$ mkdir examples
$ printf "def x = 10;\n\nx = 10;">examples/ex1.pir
```

Notice the file ends with '.pir', the standard extension for VampIR files. To compile this into a PLONK circuit, we must first set up public parameters.

```
$ target/debug/vamp-ir setup -o examples/params.pp

> * Setting up public parameters...
> * Public parameter setup success!
```

This will create the file ‘params.pp’ within our ‘examples’ directory. The `-o` argument indicates an output file and is equivalent to `--output`. We can now create the circuit associated with our file.

```
$ target/debug/vamp-ir compile -u examples/params.pp \
                               -s examples/ex1.pir \
                               -o examples/circuit.plonk

> * Compiling constraints...
> ** Inferring types...
> x[2]: int
> * Reading public parameters...
> * Synthesizing arithmetic circuit...
> * Serializing circuit to storage...
> * Constraint compilation success!
```

This will create our compiled circuit in the file ‘circuit.plonk’ within the ‘examples’ directory. The `-u` argument indicates a universal parameter file and is equivalent to `--universal-params`. The `-s` argument indicates a source file and is equivalent to `--source`.

Notice that types for defined expressions are inferred during compilation. VampIR has a simple type system that is mostly implicit. This will be explained in more detail later on. In this simple example, ‘x’ is inferred to be an ‘int’, that is, an integer that will be interpreted as a field element during compilation.

We are now in a position to synthesize a zero-knowledge proof of circuit correctness.

```
$ target/debug/vamp-ir prove -u examples/params.pp \
                             -c examples/circuit.plonk \
                             -o examples/proof.plonk

> * Reading arithmetic circuit...
> * Soliciting circuit witnesses...
> * Reading public parameters...
> * Proving knowledge of witnesses...
> * Serializing proof to storage...
> * Proof generation success!
```

This will create our compiled proof in the file ‘proof.plonk’ within the ‘examples’ directory. The `-c` argument indicates a circuit file and is equivalent to `--circuit`.

The last thing we may want to do is verify the circuit.

```
$ target/debug/vamp-ir verify -u examples/params.pp \
                              -c examples/circuit.plonk \
                              -p examples/proof.plonk

> * Reading arithmetic circuit...
> * Reading zero-knowledge proof...
> * Public inputs:
> * Reading public parameters...
> * Verifying proof validity...
> * Zero-knowledge proof is valid
```

This will verify the proof we created. In this case, we’ve just verified a zero-knowledge proof that 10 = 10. The `-p` argument indicates a proof file and is equivalent to `--proof`.

Other than `help`, we have used every available VampIR command; `setup`, `compile`, `prove`, and `verify`. These commands define all current methods for interacting with VampIR. It is a very simple system.

1.3 Proof Validity and Interaction

We do not need to give values to variables upfront. If our file was instead

```
x = 10;
```

without declaring the value of 'x', VampIR would interpret this unbound variable as an input needing to be specified during proof generation. If we save this in the file 'ex2.pir' and compile it to a circuit, we can see that VampIR will ask for an input when it's needed.

```
$ printf "x = 10;">examples/ex2.pir
$ target/debug/vamp-ir compile -u examples/params.pp \
                               -s examples/ex2.pir \
                               -o examples/circuit2.plonk

> * Compiling constraints...
> [...]
> * Constraint compilation success!

$ target/debug/vamp-ir prove -u examples/params.pp \
                             -c examples/circuit2.plonk \
                             -o examples/proof2.plonk

> * Reading arithmetic circuit...
> * Soliciting circuit witnesses...
> ** x[2] (private):

$ 9

> * Reading public parameters...
> [...]
> * Proof generation success!
```

It asked for the private value for 'x', to which I input '9'. This should create an invalid proof this time as 9 does not equal 10. If we try verifying the proof, we see that it's invalid.

```
$ target/debug/vamp-ir verify -u examples/params.pp \
                              -c examples/circuit2.plonk \
                              -p examples/proof2.plonk

> * Reading arithmetic circuit...
> [...]
> * Verifying proof validity...
> * Result from verifier: Err(ProofVerificationError)
```

As you can see, we get a verification error.

2 Programming in VampIR

2.1 Basic Arithmetic

VampIR's arithmetic denotations are fairly straightforward, being very similar to what's present in many mainstream languages.

VampIR provides four different methods to denote numbers. One can use

- standard decimal notation
- hexadecimal notation, so long as the number is prefixed by `0x`
- octal notation, so long as the number is prefixed by `0o`
- binary notation, so long as the number is prefixed by `0b`

The following example program demonstrates all the number notations.

```
0xee9a592ba9a9518 = 1074572035719075096;
1074572035719075096 = 0o73515131127246512430;
0o73515131127246512430 =
0b1110111010011010010110010010101110101001101010010101000110000;
```

Notice that VampIR allows for many top-level equations in the same program.

One can add, subtract, and multiply numbers, using, respectively, the infix operators `+`, `-`, and `*`.

```
20 * 3 - 20 = 20 + 20;
```

VampIR uses a standard order of operations where multiplication takes precedence over addition and subtraction, and everything is left-associated otherwise. Further clarification can be made using parentheses.

```
1 + 2 + 3 - 4 * 5 - 6 + 7 =
((((1 + 2) + 3) - (4 * 5)) - 6) + 7;
```

VampIR also allows for division in arithmetic circuits, denoted with the infix operator `/`.

```
15/3 = 5;
```

Division has the same precedence as multiplication, and associates to the left, otherwise.

```
2/3*4/5 = ((2/3)*4)/5;
```

This operation will give an output using the division of the underlying field, meaning constraints may differ in truth value depending on the compilation target. If one uses a prime field of size
52435875175126190479447740508185965837690552500527637822603658699938581184513,
used, for example, in BLS12-381, then the following is valid.

```
2/3 = 0x26a48d1bb889d46d66689d580335f2ac713f36abaaaa1eaa5555555500000001;
```

If one tries dividing by zero, they will get an error at some point in the process depending on where the division occurs. `1/0 = 1;` will produce an error during type inference. `1/x = 1;` will produce an error if 'x' is set to 0 during witness solicitation.

VampIR also possesses a single unary operation in the form of negation, denoted with `-`. Negation must be surrounded by parentheses unless it's iterated with additional negations.

```
(-10) = 0 - 10;
0 - 10 = (---10);
```

Exponentiation is also available via the infix operator `^`.

```
2 ^ 2 = 4;
```

Exponentiation has greater precedence than any other arithmetic operator.

```
5 * 6 ^ 7 = 5 * (6 ^ 7);
```

Only constant exponents are allowed, as they can be translated into iterated multiplication at circuit compilation time. `2 ^ x = 4;` will create an error during type inference.

Exponentiation by 0 will always produce 1, even if the base is 0.

```
2 ^ 0 = 1;
0 ^ 0 = 1;
0 ^ 2 = 0;
```

[Note: this section may describe a bug instead of intended behaviour]
Exponentiation by negatives is a bit weird. They are offset by 1 from what one might expect.

```
2 ^ (-5) = 1 / 2 ^ 4;
6 ^ (-2) = 1 / 6;
5 ^ (-3) = 1 / 5 ^ 2;
```

Exponentiating by -1 will always produce 1, with the exception of 0.

```
2 ^ (-1) = 1;
6 ^ (-1) = 1;
5 ^ (-1) = 1;
0 ^ (-1) = 0;
```

2.2 Functions

The `def` construct allows one to define, not just constants, but functions as well.

```
def square x = x ^ 2;

square 4 = 16;
```

The name of the function is denoted by the first token after `def`. Function application is denoted by juxtaposition, a la some functional languages such as Haskell.
During compilation, the type checker will identify `square`'s type as a function from integers to integers.

```
> square[3]: (int -> int)
```

Functions can have as many arguments as you want.

```
def f x y z = x * z + y;

f 4 5 6 = 29;
```

The type of this function indicates multiple arguments.

```
> f[5]: (int -> (int -> (int -> int)))
```

This tells us that each argument will take in an integer and produce a function, until the third argument, after which it will produce an integer. This indicates that application is left-associative and functions are curried by default (more on this in section 2.4).

```
((f 4) 5) 6 = f 4 5 6;
```

Definitions can contain definitions inside of them.

```
def g x = {
  def k = 20;
  x * k
};

g 3 = 60;
```

Notice the usage of curly braces to define blocks that scope the function. Also, notice that the semicolon for the definition comes after the closing curly brace, and there is no semicolon for the last line before that brace.
Internal `def` statements should be used similarly to `let` statements common in many functional languages. Internal `def` statements can be functions as well.

```
def cube x = {
  def square x = x * x;
  x * square x
};

cube 3 = 27;
```

Types for internal `def` statements are not given during compilation, but they are still checked. There is no limit to the number of lines or nestings within a definition.

```
def power x = {
  def hypercube x = {
    def square x = x * x;
    square (square x)
  };
  def cube x = {
    def square x = x * x;
    x * square x
  };
  cube (hypercube x)
 };

power 2 = 4096;
```

Overlapping names are allowed. References to names will refer to the most recent binding.

```
def x = 4;
def x = 8;
x = 8;
```

During type checking, both variables are listed.

```
> x[2]: int
> x[3]: int
```

The numbers next to the names are, in some sense, the "true" names of the variables. Each variable has a unique number identifying it, allowing the system to avoid confusing variables with the same name.

Equations can also appear in definitions. While definitions **must** return a value, and so cannot contain only an equation, they can act as gates for other values.

```
def h x = {x = 4; x};

h 4;
```

This illustrates that a VampIR file does not need any top-level equations, but will still represent a proposition through the equations enforced by function calls. `h 5;` will produce an invalid proof, in this case.

Equations aren't enforced if their parent function is never actually called.

```
def j x = {0 = 1; x};

1 = 1;
```

will produce a valid proof because 'j' is never fully instantiated. Equations become part of the circuit when their parent function is fully instantiated. `def k = {0 = 1; 4};` **will** produce an invalid proof since 'k' is already fully instantiated.

Using polynomial constraints, we can create sophisticated, reusable checks. The following, for example, checks that a field element is a boolean; either 0 or 1.

```
def isBool x = {
  (x - 1) * x = 0;
  x
};

isBool 0;
isBool 1;
```

2.3 Tuples

The only first-order data structure provided in VampIR is the tuple. We can use them in place of numbers to check multiple things in parallel.

```
def xs = (1, 2);

xs = (1, 2);
```

During type checking, we see xs's type.

```
> xs[2]: (int, int)
```

This indicates that xs is a pair of integers. We can also iterate tuples as much as we want.

```
def xs = (1, 2, 3, 4, 5, 6, 7, 8, 9);

xs = (1, 2, 3, 4, 5, 6, 7, 8, 9);
```

During type checking, we observe the type

```
> xs[2]: (int, (int, (int, (int, (int, (int, (int, (int, int))))))))
```

This indicates that tuples with more than two entries are, internally, just nested pairs associated to the right.

```
(1, 2, 3) = (1, (2, 3));
```

`(1, 2, 3) = ((1, 2), 3);` will produce a type error. We cannot compare tuples to numbers, so `1 = (1, 1);` will produce an error. We also cannot treat tuples as numbers, so, for example, `(1, 2) + (3, 4);` will produce an error.

VampIR provides the ability to define functions over tuples using pattern matching.

```
def add (a, b) (x, y) = (a + x, b + y);

add (1, 2) (3, 4) = (4, 6);
```

We can define many standard tuple construction and manipulation functions.

```
def fst (x, y) = x;
def snd (x, y) = y;
def third (x, y, z) = z;
def dup x = (x, x);
def swap (x, y) = (y, x);
def assoc (x, (y, z)) = ((x, y), z);
```

As you can see by `assoc`, one can pattern match a tuple within another tuple. Additionally, the fact that n-tuples are nested pairs means that `fst` acts uniformly on tuples; using that definition we'd have `fst (1, 2, 3) = 1`.

The types here tell us something interesting. We see, for example, that;

```
> fst[4]: (([24], [25]) -> [24])
> dup[13]: ([43] -> ([43], [43]))
> swap[16]: (([50], [51]) -> ([51], [50]))
```

The numbers in square braces denote variable types. These functions do not have a fixed type, but their types can be instantiated to be different types, depending on the context. Different calls within the same program can also have different types. For example, we can have both of these equations in the same file:

```
fst (1, 2) = 1;
fst ((1, 2), 3) = (1, 2);
```

Even though we call the same function on both lines, we get different types each time. On the first line, it returns an integer while on the second it returns a pair.

VampIR also provides a 0-tuple denoted `()`, analogous to the return value of a unit type in many functional languages or a void type in many imperative languages. It's the closest thing to getting a function to return nothing.

```
def tt = ();
def f x = {
  x = 2;
  ()
};

f 2 = tt;
```

During type inference we see the following;

```
> tt[2]: ()
> f[4]: (int -> ())
```

indicating that `tt` is an element of the unit type and that `f` will return an element of the unit type on an integer input.

Using units, we can design some of our functions to act like list manipulation programs. That is if we think of `(x, y)` as `Cons x y` and `()` as `Nil`, we can encode a list like '[1, 2, 3, 4]' as `(1, 2, 3, 4, ())`. The utility of this lies in the fact that each entry will appear in the head of a pair. This allows us to define more flexible programs. Modifying a few of our previous examples, we can have

```
def swap (x, y, r) = (y, x, r);
swap (1, 2, ()) = (2, 1, ());
swap (1, 2, 3, ()) = (2, 1, 3, ());
swap (1, 2, 3, 4, ()) = (2, 1, 3, 4, ());

def tail (x, y) = y;
tail (1, 2, ()) = (2, ());
tail (1, 2, 3, ()) = (2, 3, ());
tail (1, 2, 3, 4, ()) = (2, 3, 4, ());

def total (x, y, r) = x+y;
total (1, 2, ()) = 3;
total (1, 2, 3, ()) = 3;
total (1, 2, 3, 4, ()) = 3;

def add (a, b, q) (x, y, r) = (a + x, b + y, ());
add (1, 2, ()) (3, 4, ()) = (4, 6, ());
add (1, 2, 3, 4, 5, ()) (6, 7, 8, ()) = (7, 9, ());
```

VampIR ultimately generates a fixed data structure, meaning that there can't be a proper variable-length list type. As such, many list manipulation programs that one might take for granted can't be defined in full generality in VampIR. However, this technique can recover some of that functionality in practice.

2.4 Higher-Order Functions

```
def f x y = {0 = 1; x};

f 2;
```

will not produce an invalid proof.

3 Witness Generation

How to make your circuits more fresh... [TODO]

4 Proof Backends

Addition, Multiplication, Division, Oh my! [TODO]

5 Appendix A: Cookbook

Get a load of these delicious recipes... [TODO]

References