The VampIR Book

The VampIR Team

Contents

1 What is VampIR and How is it Used? 2
- 1.1 Why VampIR? . . . 2
- 1.2 Using VampIR . . . 2
- 1.3 Proof Validity and Interaction . . . 4

2 Basic Arithmetic in VampIR 5

3 Witness Generation 6

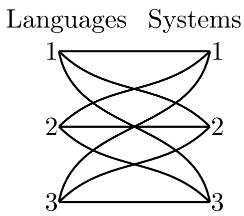
4 Proof Backends 7

5 Appendix A: Cookbook 8

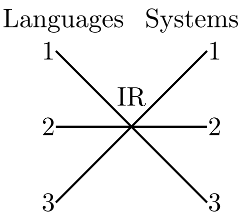
1 What is VampIR and How is it Used?

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;

Listing 1: A Very Simple VampIR File

```
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 Listing 1 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 at compile time. 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 inputed ‘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 Basic Arithmetic in VampIR

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

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