Direct circuit compilation from Rust source code comes to the public beta testing phase

After intensive testing, we're happy to announce the public release of the Rust circuit compiler built on top of the zkLLVM core. The idea behind zkLLVM is to lower the entry point for developers who want to implement zk technology into their applications. One of the ways to do that is to allow them to generate circuits directly from mainstream languages. Until now, zkLLVM enabled circuit definition from C++, with Rust being in test mode. With this last release, developers can use Rust as a source to define circuits.

Many popular zero-knowledge libraries are written in Rust using Rust cryptography libraries (arkworks, bellman, etc.), proving it to be a suitable language for the job. Same as with C++, zkLLVM will provide developers with a handful of circuit-optimized algorithms and structures in the form of an SDK.

The main goal of the zkLLVM circuit compiler [link to zkLLVM landing] is to simplify the process of circuit development while keeping it secure. Instead of manually defining circuits with some zk-specific DSLs (like Cairo, Noir, Circom, etc.), developers would benefit from using a well-known programming language like C++ or Rust. zkLLVM lets turn high-level source code directly into an arithmetic circuit, and it also partly automates the process of circuit definition, which accelerates the process speed and lowers the risk of manual mistakes in circuits.

Superior Performance without zkVMs

The crucial part of it: there is no Virtual Machine involved in the process. Instead of executing your code in a zkVM, zkLLVM circuit compiler outputs the circuit right away, and it can be immediately passed for the proof generation. On the contrary, zkVMs prove the whole state of code execution - which means additional time and costs while proving the circuit due to the circuit size. zkLLVM produces circuits that describe only the algorithm itself. This dramatically reduces the circuit size, proof generation, and verification costs. For example, an implementation of a Merkle tree outputs a circuit of a Merkle tree. Nothing else.

Moreover, zkLLVM understands algebraic types in the source code. Most of your code is based on some algebraic types of implementation. With the help of zkLLVM built-in types and functions circuit compiler will directly operate on Galois fields or elliptic curves, making the resulting circuit even smaller. We added a number of commonly used algebraic primitives and hashing functions.

zkLLVM toolchain provides a full-cycle circuit development environment

Before we jump to the step-by-step description, let's break down the main components of circuit definition. There are two initial parts to PLONK circuits:

- Constraint system

, which is a set of gates representing the algorithm and its constraints on the data. The set of constraints remains the same no matter what data inputs are used. It can be generated once and used many times to prove computations.

- Execution trace

, a set of internal variables assignments, represents the dataset the algorithm works with. This part requires to actually execute the code. In zkLLVM, there is a lightweight circuit unpacker called Assigner

that is used for that purpose.

With this in mind, here's a plan on how to use =nil; toolchain for a full-cycle circuit development from Rust source code.

Once you have your algorithm described in Rust, you then use Compiler and Assigner to get the circuit itself and the execution trace. For the latter, you pass the output of the Compiler to the Assigner. You can also verify your proof locally. Similar to development in general, you may need some tools to debug your code. The great thing here is that because zkLLVM is a compiler, you can use the same debugging approach as with normal code to debug circuit generation and assignment. The final destination point is the Proof Market, where you upload your circuit and get your proof generated by one of the provers from the distributed outsource provers' network.

How to run Rust circuit compiler

As a Rust developer, you may be familiar with cross-compilation features.Rustup is a standard tool that manages toolchains for you, and if you followed the default Rust installation process, you likely already have it. Rust circuit compiler may be installed simply as another toolchain for your existing compiler (see the Installation guide for instructions). This will allow you to seamlessly use zkLLVM's Rust frontend, standard library, and Cargo just by specifying the toolchain name:

cargo +zkllvm build

As was mentioned earlier, the Assigner will need to interpret the output of the Compiler. That's why you need to specify the compilation target:

cargo +zkllvm build --target assigner-unknown-unknown

Compiler and Assigner use LLVM assembly code representation to interact. That means that Compiler will generate LLVM assembly file, which you will pass the the Assigner. These files are generated both from C++ and Rust compiler.

Code Examples

So, how do you define a circuit function? Here goes an example of such one:

[circuit]

fn my_circuit_function() {}

[circuit]

attribute marks the entry point of the implemented circuit algorithm. Now after you pass the output to the Assigner, it will recognize my_circuit_function

as a circuit function and generate a circuit for it.

Let's look at the simple example of a circuit function:

[circuit]

fn my_circuit_function(a: PallasBase, b: PallasBase) -> PallasBase { (a + b) * a + b * (a - b) * (a + b) }

Compiler will transform this source code to LLVM assembly representation:

; Function Attrs: circuit uwtable __zkllvm_field_pallas_base @my_circuit_function(__zkllvm_field_pallas_base %a, __zkllvm_field_pallas_base %b) { start: %_4 = add __zkllvm_field_pallas_base %a, %b %_3 = mul __zkllvm_field_pallas_base %_4, %a %_11 = sub __zkllvm_field_pallas_base %a, %b %_9 = mul __zkllvm_field_pallas_base %b, %_11 %_8 = mul __zkllvm_field_pallas_base %_9, %_4 %0 = add __zkllvm_field_pallas_base %_3, %_8 ret __zkllvm_field_pallas_base %0 }

After passing this to the Assigner, you will get a 2KB circuit file and assignment table. Upload them to Proof Market and get your proof. That's it!

zkLLVM's Rust SDK is arkworks. Superoptimizations.

While being able to compile arbitrary Rust code to circuits, zkLLVM also enables developers to superoptimize the circuit using specific SDK builtins. That's why the crucial part of it is a solid and flexible SDK. All the complications will be hidden behind the library which provides developers with cryptographic primitives. As a result, devs can write code that can be compiled and run on their local machine natively, as well as easily compiled into a circuit form.

Each frontend has its own SDK: zkLLVM's C++ frontend uses Crypto3 cryptography suite as an SDK.

Rust frontend's SDK begins with arkworks. It's a rather popular library and well-optimized for performance. But there are no limits here: any library may be adapted for circuit compiler rather easily or being compiled without specific superoptimizations.

Just like with our C++ SDK, you can build your own zero-knowledge applications using circuit-optimized common algorithms:

- Proof systems
- Commitment schemes
- Hashes and cyphers
- Signatures
- Marshalling and serialization

Build your zk-application in pure Rust

Armed with the Rust circuit compiler and the arkworks SDK, you can:

- Build a state-proof- of consensus-proof-based zk-Bridge from any protocol written in Rust to EVM (we told more about

our in-EVM verifier in previous blog posts)

- Create a zkRollup from Rust implementation of a state machine/VM. For example, a Rust implementation of the EVM can be used to build a zkEVM
- Implement a zk-version of your favorite game (see our efforts onzk-DOOM)
- Build a production-ready zk-application in Rust using native integration with theproof-market.

Thanks for reading. If you have any questions left, don't hesitate to follow us and discuss on any platform you prefer.

- Discord
- Telegram
- Twitter