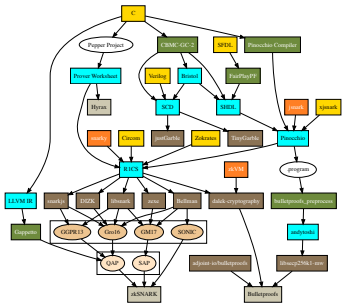
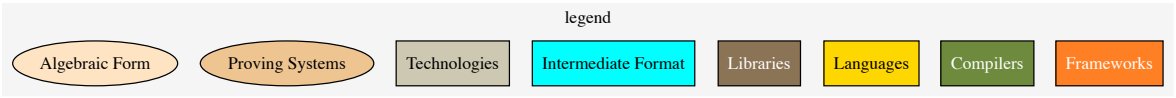
HLL

High-Level languages for zkSNARKs, Fully-Homomorphic-Encryption and Multi-Party-Computation





zkSNARKs

High-Level Languages for zkSNARKs

- https://github.com/arnaucube/go-snark
- https://github.com/republicprotocol/zksnark-rs
- https://github.com/o1-labs/snarky
- Pinocchio & Gappetto - https://github.com/corda/msr-vc
- Jsnark - https://github.com/akosba/jsnark
- xjsnark - https://github.com/akosba/xjsnark
- Pepper Project - https://github.com/pepper-project/
- Circom - https://github.com/iden3/circom
- ZoKrates - https://github.com/Zokrates/ZoKrates

Forms

QAP

- https://www.di.ens.fr/~nitulesc/files/slides/QAP.pdf
- https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649

Proving Systems

- Gro16
- SONIC
- GGPR13
- GM17

Technologies

Hyrax

Hyrax is a doubly-efficient (meaning, for both the prover and the verifier) zkSNARK which requires no trusted setup.

The source-code is available at: https://github.com/hyraxZK/hyraxZK

Hyrax uses Prover Worksheet as an intermediate form.

zkSNARKs

- https://blog.coinfabrik.com/wp-content/uploads/2018/12/zkSNARKexplained.pdf

Bulletproofs

https://eprint.iacr.org/2017/1066.pdf

https://crypto.stanford.edu/bulletproofs/

> *Bulletproofs are short non-interactive zero-knowledge proofs that require no trusted setup. A bulletproof can be used to convince a verifier that an encrypted plaintext is well formed. For example, prove that an encrypted number is in a given range, without revealing anything else about the number. Compared to SNARKs, Bulletproofs require no trusted setup. However, verifying a bulletproof is more time consuming than verifying a SNARK proof.*

Implemented in the following libraries:

- adjoint-io/bulletproofs

- dalek-cryptography
- libsecp256k1

Proving Systems

SONIC

https://github.com/zknuckles/sonic

Gro16

https://eprint.iacr.org/2016/260.pdf

GGPR13

- Quadratic Span Programs and Succinct NIZKs without PCPs

GM17

- Snarky Signatures: Minimal Signatures of Knowledge from Simulation-Extractable SNARKs

Libraries

adjoint-io/bulletproofs

A Haskell library from Adjoint.io which implements range proofs, inner product range proofs, aggregate logarithmic proofs and subsequently arithmetic circuits using Bulletproofs.

- Source code on GitHub
- Apache License, 2.0

dalek-cryptography

> *A pure-Rust implementation of Bulletproofs using Ristretto.*
>
> *The fastest Bulletproofs implementation ever, featuring single and aggregated range proofs, strongly-typed multiparty computation, and a programmable constraint system API for proving arbitrary statements (under development).*
>
> *This library implements Bulletproofs using Ristretto, using the ristretto255 implementation in [curve25519-dalek](https://doc.dalek.rs/curve25519_dalek/index.html). When using the parallel formulas in the curve25519-dalek AVX2 backend, it can verify 64-bit rangeproofs*

> ***approximately twice as fast*** *as the original libsecp256k1-based Bulletproofs implementation.*

- Source code
- MIT License

Compilers

CBMC-GC-2

> *CBMC-GC is a compiler for C programs in the context of secure two-party computation (STC). It compiles a C program that specifies a secure computation into a circuit which can be read in by an STC platform, which then performs the secure computation between two parties A and B:*

Toolchain Overview

- Source code

Languages

C

Compiled By:

- Pepper Project
- CBMC-GC

SFDL

The 'Secure Function Definition Language' was defined by the FairPlay MPC project, it is compiled into the intermediate-form 'SHDL' - the 'Secure Hardware Definition Language' - where statements are decomposed into single-static-assignment circuit form operating on individual bits. The FairPlay compiler is not only well suited for MPC, but ebcause it compiles down to operations on individual bits it's also well suited for general garbled circuits and fully-homomorphic-encryption.

Examples

```
program And {
        type Byte = Int<8>;
        type AliceInput = Byte;
        type BobInput = Byte;
```

```
        type AliceOutput = Byte;
        type BobOutput = Byte;
        type Input = struct {AliceInput alice,  BobInput bob};
        type Output = struct {AliceOutput alice, BobOutput bob};

        function Output output(Input input) {
           output.alice = (input.bob & input.alice);
           output.bob = (input.bob & input.alice);
        }
}
```

- And.sfdl
- Billionaires.sfdl
- KDS.sfdl
- Median.sfdl
- Millionaires.sfdl

Circom

Verilog

Compiled By

- Synopsys Design Compiler
- Yosys-ABC synthesis tools

The circuit_synthesis repository by TinyGarble includes a toolchain for compiling Verilog to netlist form supported by TinyGarble. This repository includes many examples.

Converted To

- v_2_scd.cpp, Netlist -> SCD

SCDL

> *Simple Circuit Description Language - simple functional language to describe circuits with a very basic implementation of a “compiler”.*

- https://github.com/ciphron/scdl

Using single-line expressions in a somewhat functional style this semi-high-level language bridges the gap between high-level and low-level.

Example:

```
include "base.scdl"

input A : 2
input B : 2

func gt(X : 2, Y : 2) = \
  or(X[1]*not(Y[1]), eq(X[1], Y[1])*X[0]*not(Y[0]))

func out = gt(A, B)
```

Intermediate Formats

Intermediate forms are simplified low-level representations which require little to no complex parsing.

Pinocchio

https://github.com/Ethsnarks/ethsnarks-il/tree/master/cxx

Prover Worksheet

SHDL

andytoshi

SCD

- https://github.com/irdan/justGarble/blob/master/scd/SCD_Format

> *An SCD file contains a compact description of a boolean circuit or a boolean circuit topology (a description with gate types are unspecified). Circuits are modeled closely after Foundations of garbled circuits by Bellare, Hoang and Roagaway, available from eprint.iacr.org. Here a circuit is a 6-tuple f = (n, m, q, A, B, G, O) where n, m, and q are integers representing the number of inputs, outputs, and gates in the circuit.*

> *Consider a simple circuit implementing f(a_1,a_2,a_3) = (a_1 & a_2) | a_3. There are three inputs, one output, two gates, and five logical wires. We will add the extra dummy wire to make it six wires overall. An SCD representation of this circuit could be as follows: (3, 1, 2, [1,4], [2,3], [8, 14], [5]), indicating that there are 3 inputs, 1 output and 2 gates, the first inputs to the gates are 1 and 4, the second inputs are 2 and 3, the gate types are 8 = 1000 = AND, and 14 = 1110 = OR, and the output of the circuit is wire 5.*

Bristol

- https://homes.esat.kuleuven.be/~nsmart/MPC/

This file format is suited for the representation of sequential binary circuits, where each wire represents a single bit.

To understand the format, each file consists of:

- A line defining the number of gates and then the number of wires in the circuit.
- Then two numbers defining the number n_1 and n_2 of wires in the inputs to the function given by the circuit.
  - We assume the function has at most two inputs; since most of our examples do. If the function has only one input then the second inputs size is set to zero.
- Then on the same line comes the number of wires in the output n_3.
- The wires are ordered so that the first n_1 wires correspond to the first input value, the next n_2 wires correspond to the second input value. The last n_3 wires correspond to the output of the circuit.
- After this the gates are listed in the format:
  - Number input wires
  - Number output wires
  - List of input wires
  - List of output wires
  - Gate operation (XOR, AND or INV)

So for example:

```
2 1 3 4 5 XOR
```

corresponds to:

```
w5 = XOR(w3, w4)
```

Examples:

- AES-non-expanded
- AES-expanded
- DES-non-expanded
- DES-expanded
- md5

- sha-1
- sha-256
- adder_32bit
- adder_64bit
- mult_32x32
- comparator_32bit_signed_lteq
- comparator_32bit_signed_lt
- comparator_32bit_unsigned_lteq
- comparator_32bit_unsigned_lt

Converts to

- SHDL
- SCD