Contributions to Rust and C++ Cryptographic Libraries for zkSNARKS

Solomon Joseph

Technical Report No. UCB/EECS-2022-252
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-252.html

December 1, 2022

Acknowledgement

I would like to oer my sincerest gratitude and thanks to Alessandro Chiesa, Dev Ojha, and Pratyush Mishra for their guiding efforts and assistance in my pursuit of research.

Contributions to Rust and C++ Cryptographic Libraries for zkSNARKS

by

Solomon Joseph

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Alessandro Chiesa, Chair
Dawn Song

Spring 2022

The thesis of Solomon Joseph, titled Contributions to Rust and C++ Cryptographic Libraries for zkSNARKS, is approved:

Chair	Alessandro Chiesa [illegible]	Date	2022.05.09
	Dawn Song [illegible]	Date	5/11/2022
		Date	

University of California, Berkeley

Contributions to Rust and C++ Cryptographic Libraries for zkSNARKS

Abstract

Contributions to Rust and C++ Cryptographic Libraries for zkSNARKS

by

Solomon Joseph

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Alessandro Chiesa, Chair

Zero-knowledge proofs [5] are a class of probabilistic proof systems that involve a powerful prover proving a theorem to a probabilistic polynomial-time verifier without revealing any information beyond the validity of the theorem to the verifier. In addition, zk-SNARKS are a class of cryptographic protocols through which a prover can efficiently prove possession of something to a verifier in a non-interactive manner. The exponential adoption of blockchain technology, among other things, has influenced a demand for high-performance cryptographic libraries providing support for zero knowledge. Chiesa's SCIPR lab provides a suite of open-source libraries, written in Rust and C++, that maintain and optimize popular cryptographic and zero knowledge implementations. In this report, I will provide an description of the libraries with which I interacted with, followed by a comprehensive review of my contributions to these libraries, namely library maintenance and algorithmic improvements.

Contents

Acknowledgments

I would like to offer my sincerest gratitude and thanks to Alessandro Chiesa, Dev Ojha, and Pratyush Mishra for their guiding efforts and assistance in my pursuit of research.

Chapter 1

Overview of Libraries

1.1 libff

libff is a C++ library for various finite fields and elliptic curve constructions. It provides optimized computation over prime $\mathbb{F}_p$, as well as even and odd field extensions (eg. $\mathbb{F}_{p^2}$, $\mathbb{F}_{p^3}$). The fields are used in various elliptic curves with fixed parameters, including Edwards, Barreto-Naehrig, and a Barreto-Naehrig implementation without dynamic code generation. My contributions to this library include migration of CI to Github Actions, adding linting support, and optimizing big-integer modular multiplication using Montgomery reduction.

1.2 arkworks-rs/algebra

Arkworks is a suite of libraries in Rust that provide a comprehensive Rust ecosystem for programming with zkSNARKS. Arkworks provides several libraries for implementing zkSNARK applications, from curves, to finite fields, to R1CS constraints.

In particular, the algebra library provides implementations of several key algebraic components that form the basis of zkSNARKS: finite fields, elliptic curves, and polynomials. The algebra library provides the following Rust crates:

1. `ark-ff` provides generic implementations of various finite fields.

2. `ark-ec` provides generic implementations for various elliptic curves, together with associated pairings.

3. `ark-poly` implements univariate, multivariate, and multilinear polynomials, as well as FFT's over finite fields.

4. `ark-serialize` provides efficient serialization and point compression for both finite fields and elliptic curves.

Chapter 2

Optimizations to C++ Libraries (libff)

2.1 Linear-Time Prover for Sumcheck Protocol

Introduction

The sumcheck protocol, originally proposed by Lund et al. [6] is an extremely powerful construction for an efficient interactive proof system. From a motivation standpoint, one can consider the delegation of computation from a computationally-weak verifier V to an untrusted and computationally-unbounded prover P. We would like to construct an efficient proof system such that:

- V does not do much more work than simply reading the input, which is linear to the input size.
- P does not do much more work than simply computing the result of the problem itself.

In 2013, Justin Thaler [7] proposed a modification to the Sumcheck Protocol to achieve a linear Prover runtime. This algorithm, which only works for multilinear functions, utilizes a bookkeeping table and multilinear extensions to provide a nontrivial prover speedup to the original sumcheck protocol, at the cost of increased space complexity.

The prototyping and benchmarking of the modified sumcheck protocol laid the groundwork for implementation in the arkworks suite of libraries.

Sumcheck Protocol

Before discussing Thaler's modification to the sumcheck protocol, we will begin by laying out the sumcheck protocol. We can define the sumcheck protocol as follows:

Definition

Let $\mathbb{F}$ be a field, H a subset of $\mathbb{F}$, n be the number of variables, and $\mu \in \mathbb{F}$ be a claimed sum. The sumcheck protocol is an interactive proof that, given oracle access to a polynomial p, determine whether $\sum_{H^n} p = \mu$.

Protocol

The sumcheck protocol consists of n rounds. At each round, the prover sends a univariate polynomial which includes a partial sum of the polynomials that have not yet been fixed by verifier randomness. Meanwhile, the verifier sends a random element at each round, implying that the sumcheck protocol is public coin. At the end of the protocol, the verifier queries the original polynomial p over the randomness it has chosen and checks that the polynomial is consistent with the one given by the prover. The scheme is detailed below.

- In the i-th round, the prover sends a univariate polynomial $p_i \in \mathbb{F}$, which in the case of the honest prover equals the partial sum

$$\phi_i(X) = \sum_{b_{i+1},\ldots,b_n} p(r_1, \ldots, r_{i-1}, X, b_{i+1}, \ldots, b_n)$$

- In the i-th round, the Verifier sends random element $r_i \in \mathbb{F}$. The Verifier also checks to see that $\sum_{x \in H} \phi_i(X) = \phi_{i-1}(r_{i-1})$
- In the final round, the verifier uses its query access to query $p(r_1, \ldots, r_n)$ and checks that $p(r_1, \ldots, r_n) = \phi_n(r_n)$. The Verifier accepts if this is the case, and rejects otherwise.

Cost Analysis

- The total communication cost of the sumcheck protocol is $O(n \cdot ideg(p))$ field elements

  Proof: In the sumcheck protocol, the prover sends n polynomials, each of degree *ideg(p)* field elements. Each of these polynomials can be specified by $ideg(p) + 1$ evaluations of the polynomial p. Meanwhile the verifier sends n field elements.

- The Verifier performs $O(n \cdot |H| \cdot ideg(p))$ field operations.

  Proof: The verifier sums p_i over H in each of the n rounds, to ensure that the prover is being honest. Thus, this amounts to $O(n \cdot |H| \cdot ideg(p))$ field operations.

- The Prover performs $O(|H|^n \cdot ideg(p) \cdot |p|)$, where $|p|$ is the cost of evaluating the multivariate polynomial p at any single location.

Proof: At the i-th round, the Prover must evaluate $|H|^{n-i}$ terms, each of which takes $|p|$ time. The Prover must also send $ideg(p) + 1$ evaluations of the polynomial p to the Verifier.

Linear-Time Sumcheck

Now that we have laid the groundwork for the sumcheck protocol, we will now visit Thaler's linear sumcheck protocol.

Multilinear Extension Lemma

Lemma: Let $f : 0, 1^n \to \mathbb{F}$. Then there is a unique multilinear polynomial $\tilde{f}$ over $\mathbb{F}$ such that $\tilde{f}(x) = f(x) \quad \forall x \in 0, 1^n$. $\tilde{f}$ is the so-called *multilinear extension* of f. In addition, given as input a list of all 2^n evaluations of f, there is an algorithm to evaluate $f(r)$ in $O(2^n)$ time.

The Multilinear Extension Lemma is used by the modification to the sumcheck protocol to provide evaluations of f over H, given the evaluations of f over the binary field $\{0, 1\}$.

High-Level Overview

Now that we have provided the requisite knowledge, we can now discuss the key points of the algorithm and their rationale, before diving deeper into the specifics and pseudocode.

First of all, since the function is multilinear, $ideg(p) = O(1)$, and each polynomial can be represented by two points. Thus, the prover sends both evaluations at each round, and the verifier is able to sum over $|H|$ by reconstructing the polynomial from the two points.

In addition, the prover is provided with a bookkeeping table, which, at rounds i, has 2^{l-i+1} entries storing the values

$$f(r_1, \ldots, r_{i-1}, b_i, b_{i+1}, \ldots, b_l) \quad \forall b_i, \ldots, b_l \in 0, 1$$

The bookkeeping table allows for a beautiful recurrence relation in the algorithm, as, in each round, $A[b] = A[b] \cdot (1 - r_i) + A[b + 2^{l-i}] \cdot r_i$

Linear-Time Sumcheck Algorithm

Algorithm 1 FunctionEvaluations($f, \mathbf{A}, r_1, \ldots, r_l$)

Input: Multilinear f on l variables, initial bookkeeping table $\mathbf{A}$, random $r_1, \ldots, r_l$;
Output: All function evaluations $f(r_1, \ldots, r_{i-1}, b_{i+1}, \ldots, b_l)$

```
1: for $i = 1, 2, \ldots$ do
2:     for $b \in 0, 1^{l-i}$ do
3:         for $t \in H$ do
4:             Let $f(r_1, \ldots, r_{i-1}, b) = A[b] \cdot (1-t) + A[b + 2^{l-i}] \cdot t$
5:         end for
6:         $A[b] = A[b] \cdot (1 - r_i) + A[b + 2^{l-i}] \cdot r_i$
7:     end for
8:     Let $\mathbf{F}$ contain all function evaluations $f(.)$ computed in line 4
9: end for
```

This algorithm is used to frontload all function evaluations of the algorithm and avoid recomputation. These evaluations are then passed into the original sumcheck protocol algorithm.

Note that since b can be used as both a number and its binary representation, we can use b elegantly as an index to the bookkeeping table. In addition, we utilize Lagrange polynomial interpolation to arrive at the multilinear extension $\tilde{f}$, which allows us to evaluate the function over H when only given evaluations over all possible bitstrings in the bookkeeping table.

Cost Analysis

Note that the communication complexity and verifier runtime of the linear-time sumcheck protocol is the same as that of the original sumcheck protocol. However, we can claim the following about the prover runtime:

- The prover runtime for the linear sumcheck protocol is $O(|H|^n)$.

  Proof: The prover now frontloads the computation of polynomial evaluations, and the above algorithm takes $O(2^n)$ time. The runtime of the prover is now dominated by the summation of the evaluations of f over b, which, in round i, is the summation of $|H|^{n-i}$ terms.

When analyzing cost, one must also acknowledge the associated space complexity that the linear sumcheck protocol adds in order to achieve a linear prover runtime. In particular, the space complexity of the algorithm is $O(2^n)$, since the prover must store the bookkeeping table. In some use cases, this space complexity will not work; thus, we must consider these tradeoffs when discussing such interactive proof schemes.

Linear Sumcheck for Products of Multilinear Functions

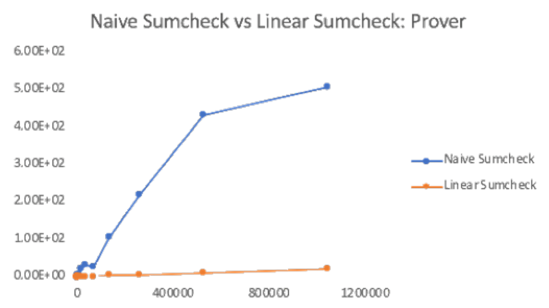
The linear-time sumcheck, enumerated in the previous subsections, also can be extended to a product of two multilinear functions. Though the proudct two functions f and g, defined in n variables, is not multilinear, the prover sends in each round:

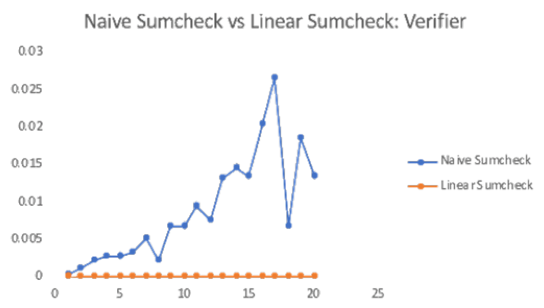
$$\sum_{b_{i+1},\ldots,b_l \in 0,1} f(r_1,\ldots,r_{i-1},b_{i+1},\ldots,b_l)\cdot g(r_1,\ldots,r_{i-1},t,b_{i-1},\ldots,b_l)$$

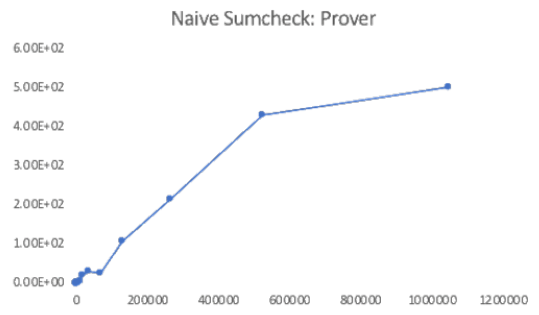
We can compute this by computing evaluations for f and g separately using the `FunctionEvaluations` algorithm and multiplying these evaluations in the original sumcheck algorithm.

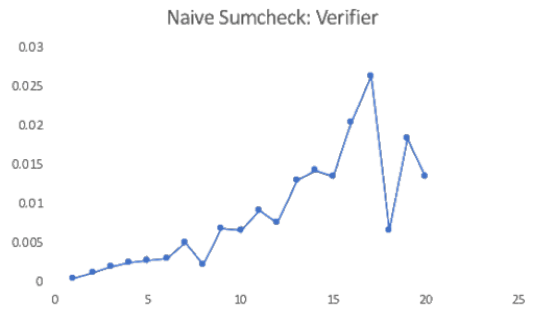
Benchmarking Speedup of Linear Time Sumcheck

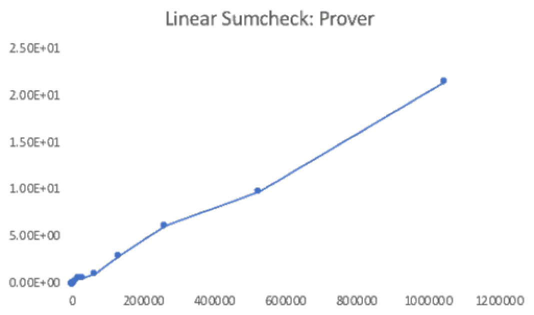
Using Thaler's algorithm and the original sumcheck protocol, I wrote a Python script to simulate a sumcheck protocol interactive proof and benchmarked the runtimes of both the verifier and prover in each version of the protocol. The results can be examined and extrapolated from the graph below:

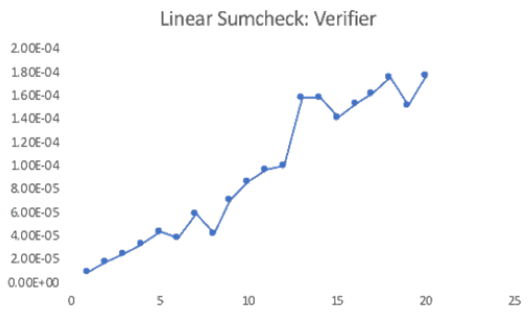












As you can see qualitatively from the graph, the prover runtime graph is quasilinear in the case of the original sumcheck, and the prover runtime is linear in the linear sumcheck case.

2.2 Faster big-integer modular multiplication using Montgomery Multiplication

A core underpinning of the `libff` library in constructing various fields and curves is the implementation of big integers and their arithmetic functions. In fact, big-integer modular multiplication is pervasive in elliptic curve cryptography, RSA cryptography, and Zero Knowledge Proofs, often invoked billions of times within a single execution of one of these protocols. As a result, micro-optimizations to the runtime of big-integer modular multiplication can provide significant overall speedups.

In particular, the developers at Consensys made available their faster big-integer modular multiplication algorithm, as implemented in their gnark library [4]. This algorithm presents a 10-15% speed improvement over existing libraries and became a suitable candidate for implementation in `libff`.

Montgomery multiplication

The modular multiplication problem involves computing the result of $a * b \bmod q$, where a, b, q are all big integers. The naive solution would be to simply multiply the two numbers and then divide the modulo, but division is extremely costly.

The way to avoid the costly division in modular multiplication is to use Montgomery Multiplication [3], in which one computes $a * b * R^{-1} \bmod q$ in lieu of $a * b \bmod q$, where R is called the *Montgomery radix*. Then, integers a and b are converted to $\bar{a} = aR \bmod q$ and $\bar{b} = bR \bmod q$, such that $\bar{a} * \bar{b} = abR$, preserving Montgomery form.

Gnark's optimization

Gnark's optimization to the CIOS Montgomery multiplication [1] saves $5N + 2$ additions, where N is the number of machine words needed to store the modulus q, and can be used whenever the highest bit of q is 0 and not all the remaining bits are set.

The final (optimized CIOS Montgomery Multiplication) algorithm is as follows:

Optimized CIOS Montgomery Multiplication

Algorithm 2 MontgomeryMultiplication(N, D, a, b, q)

Input: N : Number of machine words needed to store the modulus q, D : Word size (eg. 2^{64} on 64-bit architecture, integers a, b, and q.

Output: $a * b \bmod q$

```
1:  for $i = 1, 2, \ldots, N-1$ do
2:      (A, t[0]) := t[0] + a[0]*b[i]
3:      m = t[0]*q'[0] mod W
4:      C, _ = t[0] + m*q[0]
5:      for $j = 1, 2, \ldots, N-1$ do
6:          (A,t[j]) = t[j] + a[j]*b[j] + A
7:          (C, t[j-1]) = t[j] + m*q[j] + C
8:      end for
9:      t[N-1] = C + A
10: end for
```

A couple notes on notation:

1. $a[i]$, $b[i]$, $c[i]$ represents the i'th word of the numbers a, b, and q.
2. $q'[0]$ is the lowest word of the number $-q^{-1} \bmod R$. This number can be precomputed.
3. t is a temporary array of size $N + 2$
4. (C, S) refers to the (hi-bits, low-bits) of a two-word number.

This optimization was implemented in the `libff` library for more efficient big integer multiplication.

2.3 Linting with Clang Tidy and Github Actions

An extremely critical aspect of properly maintaining open-source libraries is the implementation of proper scaffolding and DevOps tools to ensure proper and consistent code quality. One of my tasks was incorporating a proper linting check for the `libff` library as part of the CI/CD pipeline.

The linter I incorporated was clang-tidy, a C++ linting tool that flags stylistic and logical errors in code using static analysis. After implementing the linting tool, I wrote a Github Actions script to be run on commit, ensuring proper code quality of the code commit. Finally, I retroactively fixed any linting errors in code that had been added to the library.

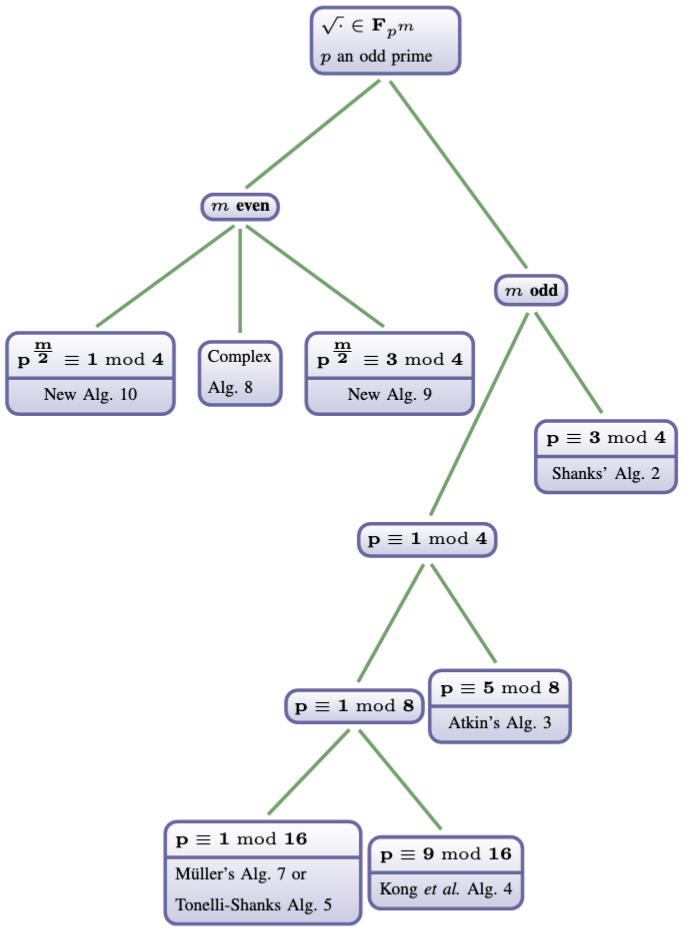
Chapter 3

Optimizations to Rust Libraries (arkworks/algebra)

3.1 More efficient square roots in extension fields

Taking square roots over finite fields is pervasive in elliptic-curve based cryptosystems: hashing a message to a point on an elliptic curve, point compression, and point counting over elliptic curves. In particular, the `arkworks/algebra` library provides computation for square roots of field elements over finite fields. Prior to this task, all square root computations defaulted to a Tonelli-Shanks algorithm. However, Adj et al. [2] proposed two novel algorithms for even field extensions of the form $\mathbb{F}_{q^2}$, with $q = p^n$, p is an odd prime and $n \geq 1$. I implemented three efficient square root algorithms over extension fields: Kong, Atkin's, and Shanks'.

Here is an overview of the various efficient algorithms for square root, based on the nature of p:


$\sqrt{\cdot} \in \mathbf{F}_{p^m}$
p an odd prime
m even
m odd
$\mathbf{p^{\frac{m}{2}} \equiv 1 \bmod 4}$
New Alg. 10
Complex Alg. 8
$\mathbf{p^{\frac{m}{2}} \equiv 3 \bmod 4}$
New Alg. 9
$\mathbf{p \equiv 3 \bmod 4}$
Shanks' Alg. 2
$\mathbf{p \equiv 1 \bmod 4}$
$\mathbf{p \equiv 1 \bmod 8}$
$\mathbf{p \equiv 5 \bmod 8}$
Atkin's Alg. 3
$\mathbf{p \equiv 1 \bmod 16}$
Müller's Alg. 7 or Tonelli-Shanks Alg. 5
$\mathbf{p \equiv 9 \bmod 16}$
Kong *et al.* Alg. 4

Shanks' algorithm

Shanks' algorithm can be applied when computing the square root of an arbitrary quadratic residue $a \in \mathbb{F}_q$. Its computational cost is a single exponentiation and two multiplications.

Algorithm 3 Shanks's algorithm for $q \equiv 3 \pmod 4$

```
Require: $a \in \mathbb{F}_q^*$.
Ensure: If it exists, $x$ satisfying $x^2 = a$, false otherwise
1: $a_1 = a^{\frac{q-3}{4}}$
2: $a_0 = a_1(a_1 a)$
3: if $a_0 = -1$ then
4:     return false
5: end if
6: $x = a_1 a$
7: return x
```

Atkin's algorithm

Atkin's algorithm can be applied when $q \equiv 5 \pmod 8$. Its computational cost is one exponentiation, four multiplications, and two squarings.

Algorithm 4 Atkin algorithm for $q \equiv 5 \pmod 8$

```
Require: $a \in F_q$.
Ensure: If it exists, $x$ satisfying $x^2 = a$, false otherwise
 1: $t = 2^{\frac{q-5}{8}}$
 2: $a_1 = a^{\frac{q-5}{8}}$
 3: $a_0 = a_1^2(a_1 a)^2$
 4: if $a_0 = -1$ then
 5:     return false
 6: end if
 7: $b = ta_1$
 8: $i = 2(ab)b$
 9: $x = (ab)(i-1)$
10: return x
```

Kong's algorithm

Algorithm 5 Kong *et al.* algorithm for $q \equiv 9 \pmod{16}$

```
Require: $a \in \mathbb{F}_q$.
Ensure: If it exists, $x$ satisfying $x^2 = a$, false otherwise
 1: $c_0 = 1$
 2: while $c_0 = 1$ do
 3:     Select randomly $c \in \mathbb{F}_q$
 4:     $c_0 = \mathcal{X}_q(c)$
 5:     $d = c^{\frac{q-9}{8}}$
 6:     $e = c^2, t = 2^{\frac{q-9}{16}}$
 7:     $a_1 = a^{\frac{q-9}{16}}$
 8:     $a_0 = (a_1^2 a)^4$
 9:     if $a_0 = -1$ then
10:         return false
11:     end if
12:     $b = ta_1$
13:     $i = 2(ab)b$
14:     $r = i^2$
15:     if $r = -1$ then
16:         $x = (ab)(i-1)$
17:     else
18:         $u = bd$
19:         $i = 2u^2ea$
20:         $x = uca(i-1)$
21:     end if
22:     return x
```

Kong's algorithm can be applied when $q \equiv 9 \pmod{16}$, and is a generalized version of the Atkin's method. Kong's algorithm can perform the square root at a cost of one exponentiation.

3.2 Unified Documentation for BigInteger struct and traits

A major pain point of research-driven code and open source libraries is surrounding scaffolding and documentation for provided libraries. In particular, The `BigInteger` crate in the `arkworks/algebra` library had minimal supporting documentation for developers. Thus, I implemented the following improvements to the Rust library documentation:

1. Added relevant descriptors to crate roots to avoid linting errors
2. Added unit tests, doc-tests, and examples for each BigInteger method
3. Added per-crate READMEs
4. Resolved inconsistencies in APIs among various BigInteger implementations.

Bibliography

[1] Tolga Acar. *High-speed algorithms and architectures for number-theoretic cryptosystems.* Oregon State University, 1998.

[2] Gora Adj and Francisco Rodrıguez-Henrıquez. “Square root computation over even extension fields”. In: *IEEE Transactions on Computers* 63.11 (2013), pp. 2829–2841.

[3] Joppe W Bos and Peter L Montgomery. “Montgomery arithmetic from a software perspective”. In: *Cryptology ePrint Archive* (2017).

[4] Gautam Botrel, Thomas Piellard, and Gus Gutoski. *Faster big-integer modular multiplication for most moduli.* URL: `https://hackmd.io/@gnark/modular_multiplication#Montgomery-squaring`.

[5] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. “The knowledge complexity of interactive proof systems”. In: *SIAM Journal on computing* 18.1 (1989), pp. 186–208.

[6] Carsten Lund et al. “Algebraic methods for interactive proof systems”. In: *Journal of the ACM (JACM)* 39.4 (1992), pp. 859–868.

[7] Justin Thaler. “Time-optimal interactive proofs for circuit evaluation”. In: *Annual Cryptology Conference*. Springer. 2013, pp. 71–89.