

Optimizations and Improvements to Cryptographic Libraries for zkSNARKs

Alexander Wu

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-102

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-102.html>

May 14, 2021



Copyright © 2021, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I offer my sincerest gratitude to Professor Alessandro Chiesa and Dev Ojha for their assistance on this project.

Optimizations and Improvements to Cryptographic Libraries for zkSNARKs

by

Alexander Wu

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Alessandro Chiesa, Chair

Professor Raluca Ada Popa

Spring 2021

The thesis of Alexander Wu, titled Optimizations and Improvements to Cryptographic Libraries for zkSNARKs, is approved:

Chair		Date	<u>2021.05.14</u>
		Date	<u>May 13, 2021</u>
		Date	<u></u>

University of California, Berkeley

Optimizations and Improvements to Cryptographic Libraries for zkSNARKs

Copyright 2021
by
Alexander Wu

Abstract

Optimizations and Improvements to Cryptographic Libraries for zkSNARKs

by

Alexander Wu

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alessandro Chiesa, Chair

With the rapid development of the theory of probabilistic proofs, zero knowledge proofs have started to gain traction in both academic circles and industry. As such, the demand has risen for convenient, high-performance cryptographic libraries that aid in the use of zero knowledge proofs and other cryptographic protocols. In this report, I will discuss my contributions to several cryptographic libraries that fulfill this demand. I will summarize the purpose of four cryptographic libraries and describe my work to make them more ergonomic and configurable through API reworks and other improvements. Then I will describe my performance optimizations, namely the cap hash optimization for the BCS Compiler. Finally, I will point out avenues for future work on these libraries.



Contents

Contents	i
List of Figures	ii
List of Tables	iii
1 Overview of Libraries	1
1.1 libff	1
1.2 libfqfft	1
1.3 libiop	1
1.4 libsnark	3
2 Application Programming Interface Improvements	4
2.1 Migration of Binary Finite Fields	4
2.2 Unified Field APIs and Tests	5
2.3 Other Miscellaneous Improvements	5
3 Performance and Algorithm Improvements	6
3.1 Motivation for Lowering Hash Usage	6
3.2 Merkle Tree Cap Hash Optimization	6
3.3 Results	8
4 Future Work	10
Bibliography	11

List of Figures

1.1	BCS Compiler	2
3.1	Normal Merkle Tree	7
3.2	Merkle Tree with Cap Hashing	7
3.3	Estimated Complexity of a SNARK with and without Cap Hashing	8
3.4	Argument Sizes of Fractal SNARK with and without Cap Hashing	9

List of Tables

2.1 All field classes in libff and libiop	4
---	---

Acknowledgments

I offer my sincerest gratitude to Professor Alessandro Chiesa and Dev Ojha for their assistance on this project.

Chapter 1

Overview of Libraries

1.1 libff

`libff` is a util library for finite fields and elliptic curves written in the programming language C++. It supports optimized computations with the prime finite field $\mathbb{F}_p$ as well as field extensions such as $\mathbb{F}_{p^2}$ and $\mathbb{F}_{p^3}$. These fields are then used in certain elliptic curves with fixed parameters, including the Edwards Curve, Barreto-Naehrig Curves, etc. The library has functionality for testing and profiling operation counts and runtime.

In this project, binary curves such as $\mathbb{F}_{2^{256}}$ were added to `libff`, among other improvements.

1.2 libfqfft

`libfqfft` is a util library for Fast Fourier Transforms on finite fields written in C++. It supports fast multipoint polynomial evaluation, fast polynomial interpolation, and fast computation of Lagrange polynomials, in addition to multi-threading using OpenMP. The library uses `libff` as a dependency.

1.3 libiop

`libiop` is a util library for IOP-based zkSNARKs written in C++. It uses both `libff` and `libfqfft` as a dependencies. It has support for testing, profiling, and benchmarking.

The library contains infrastructure for writing IOP protocols and compiling with the BCS Compiler. It includes three example protocols, the Ligero, Aurora, and Fractal protocols, as well as their compiled SNARKs. All protocols support the Rank 1 Constraint System (R1CS) problem.



Interactive oracle proofs (IOPs) are a model for proof systems that includes aspects from both interactive proofs (IPs) and probabilistically-checkable proofs (PCPs).[\[1\]](#) An IOP contains multiple rounds, where in each round, the verifier sends the prover a message, and the prover returns an oracle to the verifier on which the verifier can perform a small number of queries. After all rounds are completed, the verifier computes on the results to either accept or reject the proof.

Using this compiler, we can construct zkSNARKs that are more efficient than PCP-based implementations. The compiler preserves the proof of knowledge and zero knowledge

properties of an IOP. The produced succinct non-interactive argument (SNARG) has the benefit of being transparent and post-quantum, while only requiring the use of lightweight symmetric cryptography. [2] In addition, it compiles holographic IOPs into preprocessing SNARGs. [3]

1.4 libsnark

`libsnark` is a util library for elliptic curve based preprocessing zkSNARKs written in C++. It uses both `libff` and `libqfft` as a dependencies. It also has support for testing, profiling, and benchmarking.

The library implements general-purpose proof systems for R1CS, Bilinear Arithmetic Circuit Satisfiability (BACS), Unitary-Square Constraint Systems (USCS), Two-input Boolean Circuit Satisfiability (TBCS), and others. It also includes “gadget libraries” for writing R1CS instances. Finally, it includes example applications that utilize these protocols.

Chapter 2

Application Programming Interface Improvements

2.1 Migration of Binary Finite Fields

Before this project, the binary fields in Table 2.1 were located in the library libiop. In order to allow other users to utilize the binary fields without having to include the entire libiop library, and so that each library can have a single focus, these fields were moved to the libff library. The libff library's structure was modified to incorporate these fields.

name	type of field	mathematical expression	previous location	new location
Fp	prime base	$\mathbb{F}_p$	libff	libff
Fp2	prime extension	$\mathbb{F}_{p^2}$	libff	libff
Fp3	prime extension	$\mathbb{F}_{p^3}$	libff	libff
Fp4	prime extension	$\mathbb{F}_{p^4}$	libff	libff
Fp6_2over3	prime extension	$\mathbb{F}_{(p^3)^2}$	libff	libff
Fp6_3over2	prime extension	$\mathbb{F}_{(p^2)^3}$	libff	libff
Fp12_2over3over2	prime extension	$\mathbb{F}_{((p^2)^3)^2}$	libff	libff
gf32	binary	$\mathbb{F}_{2^{32}}$	libiop	libff
gf64	binary	$\mathbb{F}_{2^{64}}$	libiop	libff
gf128	binary	$\mathbb{F}_{2^{128}}$	libiop	libff
gf192	binary	$\mathbb{F}_{2^{192}}$	libiop	libff
gf256	binary	$\mathbb{F}_{2^{256}}$	libiop	libff

Table 2.1: All field classes in libff and libiop

2.2 Unified Field APIs and Tests

Because the binary and non-binary fields were developed separately, their APIs were different. In order to simplify the programming interface, and possibly allow computation that generalizes to both types of fields, a consistent field API was developed. It is now clearly laid out which attributes and methods are common to all fields, which are specific to non-binary fields, and which are specific to binary fields.

Although one may wish to create a parent class that all fields inherit from in order to reduce code duplication and enforce the API, C++ class inheritance significantly hinders performance. Because the fields are performance critical for many applications, I chose to keep each class separate and thoroughly document the API instead. If one wishes to write code that generalizes to all fields, C++ templates can be used.

To enforce this API, every single attribute and method in the API is tested through Google unit tests. There are separate tests for all fields, all binary fields, and all non-binary fields.

The field utils in libiop were also moved into libff, and more thorough tests were developed for them.

2.3 Other Miscellaneous Improvements

In libff, more comprehensive tests were added for all finite fields, as well as the utils such as bigint, power, conversions between fields and bits, and random element generation. The testing methodology was re-worked. Bugs were fixed and APIs were revised to be more consistent. Field parameters were separated from the corresponding elliptic curves. The documentation was updated.

In libiop, similar improvements were made. Better debug printouts were added, for example, oracles used in the protocol were given names, and their names are printed for each round when the protocol is run. Algorithms were refactored, for example, all instances of the hashchain computation for the BCS prover and verifier were refined and extracted into a single function. Code style and documentation were improved.

After libff was modified, the libraries that use it as a dependency, namely libfqfft, libiop, and libsnark, were all updated to be compatible with the new version.

Chapter 3

Performance and Algorithm Improvements

3.1 Motivation for Lowering Hash Usage

The computation needed to run a SNARK primarily includes the algebra and hashing. In some cases, the hashing may take up a significant amount of runtime. For example, when executing the recursive Fractal SNARK protocol, hashing takes up the vast majority of computation. This is because the SNARK must be encoded in R1CS to input into another SNARK, and this can only be efficiently performed on algebraic hashes such as Poseidon, which are much slower than standard hashes like Blake2. [4](#)

In these cases, reducing the number of hashes performed in a SNARK protocol has the potential to significantly improve performance. Below, I will describe one such optimization.

3.2 Merkle Tree Cap Hash Optimization

To compile oracle-based IOPs into SNARKs, the resulting prover generates Merkle trees based on the oracles. The oracle is padded and split into a number of leaves, as can be seen in Figure [3.1](#). In this example, the oracle is of length 32, and split into 8 leaves each of size 4. A hash function, referred to as the leaf hash, is applied to each leaf and entered into the bottom-most layer of the Merkle tree. Then a two-to-one hash function, referred to as the node hash, is applied to construct a binary Merkle tree. The root is then used for further computation and given to the verifier.

To generate a membership proof to the oracle, the auxiliary hashes for all paths from the required leaves to the root are given. For example, if an IOP verifier asks for the proof for the leaves underneath (h) and (i) in Figure [3.1](#), the hashes (g), (j), and (b) are provided in the membership proof. Then the verifier would be able to calculate (h), (c), (i), (d), (a), and the root and verifier membership.

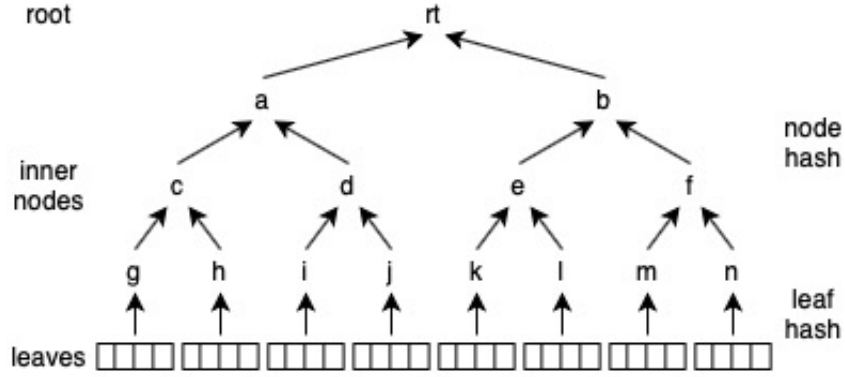


Figure 3.1: Normal Merkle Tree

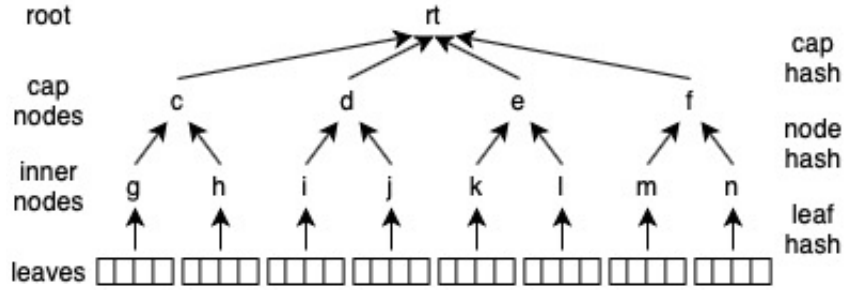


Figure 3.2: Merkle Tree with Cap Hashing

In most applications of an IOP protocol, a large number of leaves will require membership proofs (though still much less than the total number of leaves). In that case, many inner nodes in the top layers will be able to be calculated without them being in the auxiliary hashes. For example in Figure 3.1, if (h), (i), and (m) require proofs, the inner nodes (a), (b), (c), (d), and (f) can all be calculated and the only required auxiliary hash in the first three layers is (e). However, since it is a binary tree, we still need to compute every node hash to get the root. To save computation, we can instead remove some top layers and make some inner nodes direct children of the root. See Figure 3.2. In this example, the cap size is 4, but in general it can be any positive power of 2. Now instead of three hashes to compute the root from (c), (d), (e), and (f), only one hash is required. This might increase the number of auxiliary hashes required, but if there are a large number of queried leaves, the increase will be negligible.

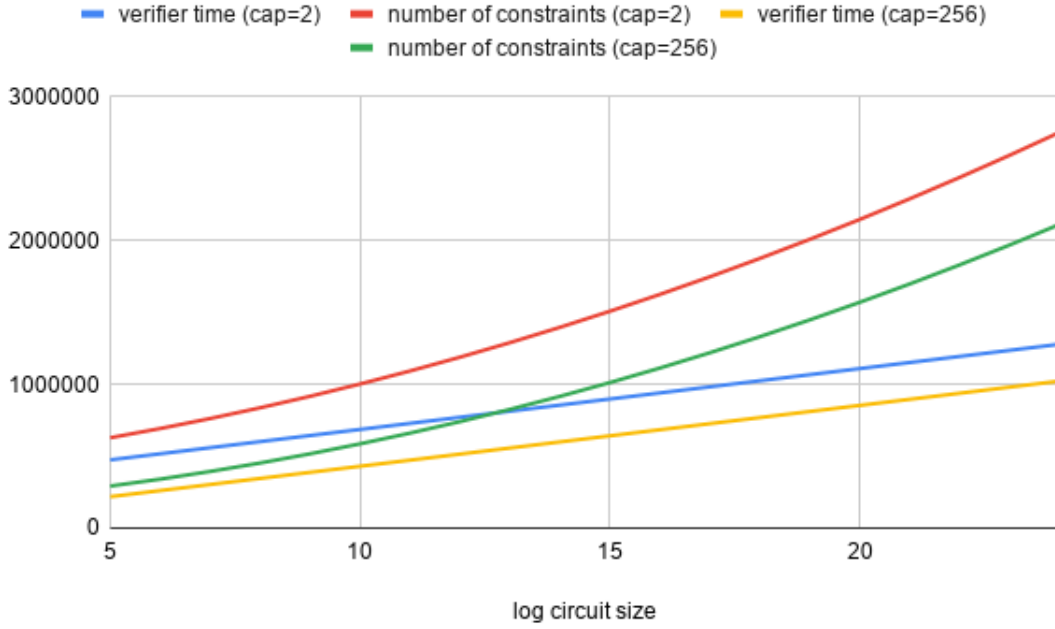


Figure 3.3: Estimated Complexity of a SNARK with and without Cap Hashing

3.3 Results

This optimization, referred to as “cap hash optimization”, was carried out on libiop’s BCS Compiler. Because the actual number of hashes calculated in a SNARK is highly random, and very large circuit sizes are required for it to exhibit the optimization benefits, I opted to perform an estimate of the time save. Figure 3.3 is a plot of the generated complexity estimates of computing on a circuit before and after the cap hash. Note that the verifier time and number of constraints do not share the same units. Both the verifier time and number of constraints are visibly lower when the Merkle tree has a cap hash with cap size 256. A low number of R1CS constraints is critical for efficient recursive SNARKs.

Although I did not show the real runtimes of a SNARK with cap hashing, I did record the total argument sizes that resulted from several executions of the Fractal SNARK protocol. Figure 3.4 shows a comparison of the actual argument sizes for an execution without cap hashing and one with a cap size of 64. It shows that cap hashing allows for a real decrease in the argument size generated by the BCS Compiler.

In conclusion, cap hashing allows a SNARK protocol to have a more efficient verifier and smaller argument size. This is particularly helpful for SNARKs where the hashes are a significant bottleneck such as recursive Fractal SNARKs.

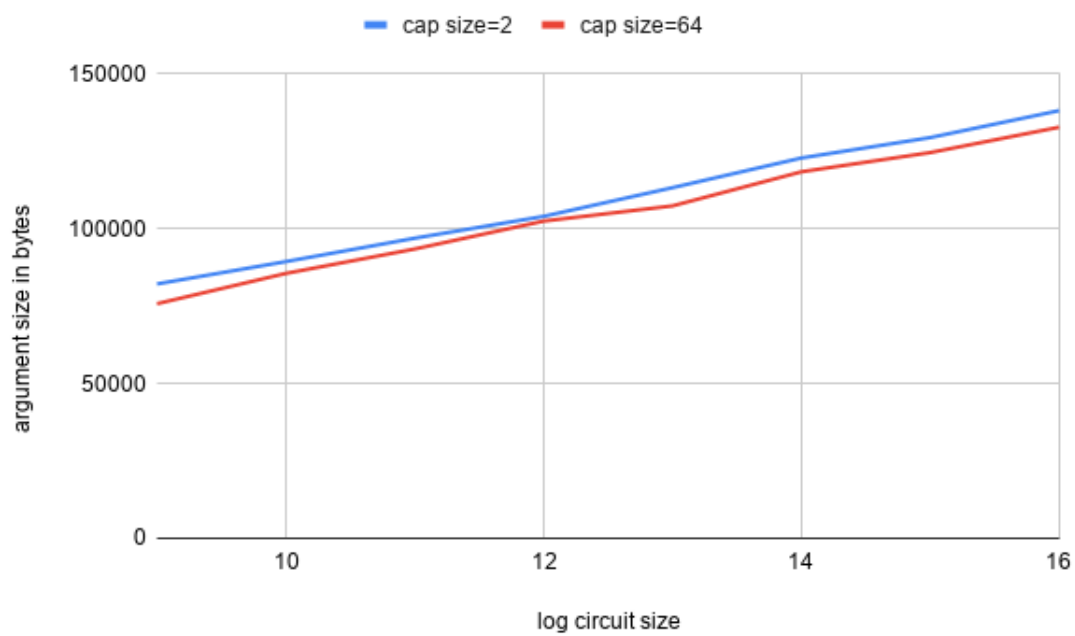


Figure 3.4: Argument Sizes of Fractal SNARK with and without Cap Hashing

Chapter 4

Future Work

Possible continuations of this project include:

- Moving the binary fields from libiop to libff is an instance of modularizing the libraries so each one is concerned with only one focus. Continuing this trend, the elliptic curves located in libff should be moved to a new library, perhaps called libecc, since as the name suggests libff is primarily concerned with finite fields. This may be slightly involved, because the non-binary fields on their own do not include the right parameters and cannot be tested unless fields are taken in from the curves.
- The fields in libff were not made to inherit from a parent field class due to performance concerns. The API was enforced through tests, but it may be worth further investigating whether there are other methods of enforcing it and possibly avoid the duplicate code, without taking a performance penalty.
- Many more minor improvements can be made to libiop. The library is an academic proof-of-concept prototype, and still requires extensive bug fixes and refactoring.

Bibliography

- [1] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. *Interactive Oracle Proofs*. Cryptology ePrint Archive, Report 2016/116. <https://eprint.iacr.org/2016/116>. 2016.
- [2] Alessandro Chiesa, Peter Manohar, and Nicholas Spooner. *Succinct Arguments in the Quantum Random Oracle Model*. Cryptology ePrint Archive, Report 2019/834. <https://eprint.iacr.org/2019/834>. 2019.
- [3] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. *Fractal: Post-Quantum and Transparent Recursive Proofs from Holography*. Cryptology ePrint Archive, Report 2019/1076. <https://eprint.iacr.org/2019/1076>. 2019.
- [4] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. *Fractal: Post-Quantum and Transparent Recursive Proofs from Holography*. Cryptology ePrint Archive, Report 2019/1076. <https://eprint.iacr.org/2019/1076>. 2019.