Implementing lattice-based cryptography in libsnark

Samir J Menon
Stanford University

Abstract

New research allows construction of SNARK's from lattice-based primitives, instead of pairings-based cryptography [BISW17]. We implement a proposed construction in libsnark, a popular existing zkSNARK library, using a quadratic arithmetic program representation and a 'crypto compiler' based on an extension of standard Regev encryption [Reg05] into a linear-only vector encryption scheme [PVW08]. Our implementation should speed up verification, which is currently unacceptably slow (40s on average hardware). Additionally, lattice-based primitives are thought to be post-quantum secure, so our zkSNARK implementation will inherit this useful property [Sim97].

1 Background & Motivation

1.1 What is a SNARK?

Using a zero-knowledge proof (ZKP), a prover can prove a statement while revealing absolutely no additional information to the verifier (that is, the verifier learns nothing besides that the statement is true).

In addition to the zero-knowledge property, we require the two standard requirements for a proof: completeness (if the statement is true, the verifier will be convinced) and soundness (a malicious prover cannot convince the honest verifier of a false statement).

We can refine our idea of a ZKP in three ways: non-interactivity, proof-of-knowledge, and succinctness.

Proof-of-Knowledge

We can require a slightly stronger notion of proof by requiring that the prover shows that she actually knows the witness. That is, in a standard zero-knowledge proof system, we only require that Peggy prove that a witness exists that satisfies the public relation; we don't require that Peggy proves that she knows this witness. In some scenarios that we'll illustrate later, it's helpful to require that Peggy actually knows some specific witness w. This is referred to as a proof-of-knowledge (PoK).

Non-interactivity

A non-interactive zero-knowledge proof (NIZK) is simply one that is completed in one step; that is, a proof π is sent by the prover to the verifier, and then the proof is complete. While these are impossible in the standard model, they are possible in the random oracle model. Using the Fiat-Shamir heuristic, we can convert any interactive proof system into a non-interactive one, using a random oracle (in practice, a cryptographic hash function) wherever in the protocol a party was supposed to generate a random value. We can also build NIZK's using the 'common reference string' (CRS) model, where everyone shares a large public string that was chosen honestly from a distribution. We come back to how CRS's help us implement these protocols in practice later.

Succinctness

In practice, the most important cost to a NIZK is the communication overhead; that is, the size of the proof π. In fact, it's relatively simple to implement a NIZK with a very large π, but this is not feasible in the real world. In particular, we say that a SNARK has quasi-optimal succinctness if the length of the proof is $\tilde{O}(\lambda)$ where λ is the security parameter and the circuit is of size 2^{λ}. The construction we've built has this property.

SNARKs

Combining PoK's and NIZKs gives us a zero-knowledge, non-interactive proof (or 'argument') of knowledge, hence the acronym "zkSNARK". We focus on building SNARK's in general, since the zero-knowledge property is easily added. These are a powerful cryptographic tool with broad range of applications.

The largest real-world deployment of zkSNARKs is Zcash, a cryptocurrency that allows users to transact entirely anonymously. It allows users to selectively make the sender, receiver, and amount entirely hidden from non-participants in the transaction.

A potentially important future application of zk-SNARKs is verifiable computing [BCGTV 2013] (Verifying Program Executions Succinctly and in Zero Knowledge). If SNARKs became less computationally expensive, we could produce feasibly short proofs of program outputs, which would have a wide range of uses in safety engineering and distributed computing.

Currently, zkSNARKs are implemented using pairing-based cryptography [SCIPR]. We look specifically at the largest currently available zkSNARK implementation, libsnark, a product of the SCIPR lab. Currently, libsnark verification times (as used in Zcash, for example) are longer than we'd like. This is primarily because, using pairing-based cryptography, we have to perform a computationally expensive group exponentiation at each step in the proof circuit.

1.2 Construction of a SNARK

We'll start by describing a linear probabilistically checkable proof (LPCP), and then we'll explain how to apply linear-only vector encryption to make the proof a SNARK.

$$\begin{array}{ccc}
\underline{\text{Prover}} & & \underline{\text{Verifier}} \\
\text{x,w} & & \text{x} \\
\pi \leftarrow P_{\text{LPCP}}(x,w) & & Q \leftarrow Q_{\text{LPCP}}() \\
 & \xleftarrow{\quad Q \quad} & \\
 & \xrightarrow{\; y = Q^T\pi \;} & \\
 & & \text{yes/no} \leftarrow D_{\text{LPCP}}(x,y)
\end{array}$$

A linear PCP is composed of three algorithms: $P(x,w)$ is used by the prover to compute a proof $\pi \in \mathbb{Z}_p^m$ given a statement $x \in \mathbb{Z}_p^\ell$ and a witness w. $Q()$ is used by the verifier to generate a query matrix $Q \in \mathbb{Z}_p^{m\times\ell}$. Finally, $D(x,y)$ tells the verifier to accept or reject the proof, where $y = Q^T\pi \in \mathbb{Z}_p^\ell$. There are existing instantiations of linear PCP's; for our purposes, it's only necessary to examine the guarantees the provide.

Essentially, a correct linear PCP guarantees that for the relation R being proven, after going to the protocol, all $(x,w) \in R$ will result in an accept, and $(x,w) \notin R$ will only result in an accept with negligible probability.

The protocol is simple. In generation, the verifier generates a query Q, independent of the statement x, and the prover generates a proof π. The verifier sends Q, the prover computes $y = Q^T\pi$, and then the verifier accepts or rejects.

The key to making this a SNARK is encrypting the query matrix Q and response $y = Q^T\pi$. We can think of this as using a 'crypto compiler' to secure the LPCP and convert into a SNARK. The verifier will now get some secret key that has been used to encrypt the query matrix, and this encrypted query matrix will now be the common reference string (we have moved the protocol to the non-interactive setting). The main requirement for the encryption system is linear homomorphism; specifically, the prover needs to be able to compute the encryption $Q^T\pi$ using only the CRS, which is the encryption of Q.

Currently, SNARKs instantiate this linearly homomorphic encryption using pairing-based cryptography, which as mentioned, can be quite slow. This has become a problem in practice; for example, libsnark verification times (as used in Zcash) are quite high (about 40 sec on standard hardware).

Instead of using pairings, we use newer lattice-based cryptographic primitives, based on the hardness of the learning with errors (LWE) problem, to build zkSNARKs that are faster to verify and quantum resistant. The speedup should come mostly from the fact that instead of doing many group exponentiations, we will be doing simpler lattice-based operations (matrix operations on 64-bit integers) which can be data-level parallelizable, perhaps through SIMD and even eventually GPUs.

2 Implementation

We use the construction described in [BISW17, Construction 4.5] to instantiate a SNARG from just a Linear PCP and a linear-only vector encryption system. We specifically use the scheme by Peikert, Vaikuntanathan, and Waters [PVW08, §7.2], which is based on the LWE problem introduced by Regev.

Libsnark uses a variety of representations of proving systems, and provides a variety of methods of converting statements in one system to statements in another. We've chosen as our starting point the rank-one constraint system, because it is broadly used and there are many other, more useful representations that can be compiled to R1CS. Each constraint in an R1CS is a triple of three vectors, A,B,C, and we can say that s satisfies that constraint if $(As)(Bs) = Cs$. For a given statement, we have many constraints, each of which has a triple (A,B,C).

We then convert these into a quadratic arithmetic program (QAP), which represents all of the constraints in just three sets of polynomials (A', B' and C'), so that the polynomials in A' evaluated at some $x = i$ give the i'th constraint's A vector, and likewise for B and C. This conversion is done using Lagrange interpolation. The reason for doing this is to enable a faster check on the constraints, as follows.

We can now, given some candidate solution s we wish to check, in one step, compute $A's \cdot B's - C's$ and check that the result is 0 at $x = 1,2,\ldots$. In fact, this is further optimized; we can actually just check that $A's \cdot B's - C's$ is divisible by $Z = (x-1)(x-2)\ldots$.

So, our actual implementation uses the system that libsnark already has to generate these polynomials from an R1CS. We then apply our 'crypto compiler', and result in a system roughly as follows:

1. In setup, we create a CRS that is the encryption of each of A', B', C'.
2. The prover, using its witness, does homomorphic operations on the CRS to create an answer $c = E()$.
3. The verifier decrypts c into s, and then checks that $A's + B's - C's = 0$ at $x = 1, 2\ldots$, using the divisibility technique described above.

For the implementing matrix operations, we use Victor Shoup's "Number Theory Library" (NTL). It has optimized implementations of matrices and vectors $\mod p$, (`mat_ZZ_p` and `vec_ZZ_p`), that we leverage.

We also started with a small piece Leo Ducas' Fully Homomorphic Encryption Library (FHEW); specifically, we used the implementation of the LWE encryption as a starting point for scalar encryption.

3 Completed work

As a proof of concept, we first changed the encryption scheme so that it used Regev encryption naively; that is, each query was for a single element in the field F_p where p is relatively small (we used $p = 1009$). While this is not secure, it allowed us to test our encryption and decryption routines, and confirm that the homomorphism is working correctly.

In the process, many of the consistency checks that were necessary in the pairing setting were no longer necessary. For example, we didn't need to check if the same coefficients were being used, because the prover couldn't change them in the encrypted CRS anyway, since the encryption scheme is semantically secure.

Next, we implemented the PVW encryption scheme. This is essentially a vectorized form of basic Regev encryption, in which we encrypt many plaintext field elements at once using a secret key that is a matrix. For our purposes, this is useful because we can now implement encryption and decryption in simple matrix operations on 64-bit integers. We tested the scheme, and ensured that we maintained all necessary homomorphic properties, in isolation from libsnark.

We then combined the PVW scheme, which was our instantiation of the linear-only vector encryption scheme required in BISW's construction, and the QAP system that libsnark provides, to create a working prototype lattice-based SNARK.

4 Results

One significant goal was to make verification times faster. We have succeeded in that end; our implementation appears to verify circuits of sizes that we have been able to test so far more quickly than the pairing-based implementation, and the verification time should be essentially constant even as we make the circuits much larger. In some small circuits, our verification times are up to 6x faster than the current ones. Our prover time seems to be on par with the current implementation, but as we scale the sizes, it appears to grow more rapidly than we'd like.

Another key question that we seek to answer in this implementation is: what parameters to this new lattice-base zkSNARK system are practical, and what kind of performance can we achieve while maintaining 128-bit security?

The parameters of interest are p, the plaintext modulus, q, the ciphertext modulus, ℓ, the number of queries, and m and n, where the encrypted matrices are $m \times n$.

For semantic security to hold, according to BISW17 Thm 4.13, we need $m \geq 3(n+\ell)\log q$. We also need $q > p^2 m$. Finally, we need n to be large enough to maintain correctness. The larger the circuit we want to verify, the larger the plaintext modulus we need, which then increases all the other parameters.

There's a significant amount of circularity in the dependencies between these parameters. We are still only able to compute circuits with at most 200 gates and 200 inputs, because we haven't found a good way to deal with very large parameters, particularly m.

We have also found that our setup time is much much larger than the pairing based implementation. While it's generally safe to disregard this as a serious penalty because the setup of the CRS is generally seen as a one-time cost, the setup is becoming so drastically expensive that it becomes difficult to practically test the implementation. So far, our analysis of why setup is so expensive has yielded only the large size of the randomly generated matrix as a source of the problem.

5 Limitations and Future work

A significant barrier to real-world use of the system is that our SNARK is not publicly verifiable yet. Adapting the scheme to the public verification setting will take significantly more work, both in the construction and implementation.

We are currently working on, and hope to complete soon, a working example of the verification of a SHA256 hash. This is a large circuit (about 20,000 gates), and we haven't yet figured out exactly how to push this prototype to the point where it can handle that size. Most importantly, the parameters we are using are insufficient, and it's not exactly clear yet how to make them bigger without introducing new issues.

We also have not tested performance in a controlled and rigorous way yet, because only very recently did we complete the fully vectorized implementation. Another next step would be systematically assessing our efforts at parameter tuning and determining what challenges still remain.

Finally, there is a large amount of work that can still be done in making the scheme more performant. Most promisingly, since most of the expensive operations are matrix-vector and matrix-matrix multiplications, there is a large amount of advanced computer hardware strategies (like using data-level parallelism/SIMD) that could make the scheme practical.

References

[BISW17] an Boneh, Yuval Ishai, Amit Sahai, and David J. Wu. Lattice-based SNARGs and their application to more efficient obfuscation. In *EUROCRYPT*, 2017.

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In *CRYPTO*, 2008.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC*, 2005.

[SCIPR] SCIPR Lab. libsnark: a C++ library for zk-SNARK proofs. Available: https://github.com/scipr-lab/libsnark. First release Jun 2, 2014.

[Sim97] Daniel R. Simon. On the power of quantum computation. *SIAM J. Comput.*, 26(5), 1997.