Implementing compact tree-hashing commitment verification in ZK-SNARK

Stefano Trevisani

May 23, 2022

1 Introduction

One of the biggest revolutions of the last decade has been the widespread adoption of blockchain-based technologies. In particular, cryptocurrencies like Bitcoin or Ethereum are widely known even among the non-crypto-enthusiasts and a huge market is growing around them. There are highly interesting applications of the blockchain even outside the financial world: in fact, anything which requires some degree of 'verifiability' in a non-trusted environment can benefit from using a blockchain. Tipically, a block of the chain does not correspond to a single transaction, as the blockchain would become too big to be stored: many transactions are inserted into a tree data-structure, called Merkle Tree (MT), which is computed bottom-up and whose root is then inserted into the blockchain. In a traditional setup (e.g. the aformentioned cryptocurrencies), all the data which is required to build a block of the blockchain is of public knowledge: when an Ethereum transaction happens, the details of that transaction are shared with the network for it to be validated. Of course, there are many scenarios were one would like the data to remain secret, as there could be risks for privacy (e.g. transactions) or for intellectual property (e.g. algorithms).

Zero-Knowledge Succint Non-interactive ARgument of Knowledge (ZK-SNARK) systems are cryptographic frameworks which allow for a party to convince other parties that he 'knows something' without revealing anything else. For example, one could convice other users that the hash of a transaction is valid without revealing the details of that transaction. ZK systems work over prime fields, and hashing algorithms like SHA-256, which are quite efficient in a 'vanilla' scenario, can become extremely slow when translated. For this reason, new hashing algorithms like MiMCHash have been designed with the ZK scenario in mind, and Augmented Binary tRee (ABR) tries to improve Merkle Trees by processing more transactions without requiring more hash function calls.

In the remainder of the report, for clarity we will always have in mind the cryptocurrency scenario, but we want to make clear that all the other use-cases are the same[1]. In Section 2, we introduce hash functions and tree-based modes of hashing. In Section 3, we introduce ZK-SNARK systems and brifely describe how they can be implemented. In Section 4, we introduce `libsnark`, a C++ library that provides many facilities to implement ZKSNARK algorithms with relative ease, and we will discuss the implementation of MiMC, MTs and ABRs using this library. In Section 5, we perform some experiments to compare MiMC with SHA, and MTs with ABRs in a ZK setting. Finally, in Section 6 we draw our conclusions and discuss possible future work directions.

[1]Up to isomorphism, of course.

2 Preliminaries

In this section we are going to introduce some fundamental concepts; while some are relatively basilar, it can still be useful to skim over them to be sure of having a firm grasp of the foundations of ZK-SNARK systems.

2.1 Computational models and complexity

A *computational model* (or model of computation) is any kind of 'device', either physical or mathematical, which is able to compute algorithms to solve problems. A particularly interesting class of problems are *decision problems*, the ones that can be answered with 'yes' or 'no'. Every computational model is able to *decide* only a subclass of all decision problems, and even then, not all can be solved *efficiently*, that is, by using an amount of resources (tipically, time and space) which is upper-bounded by some polynomial over the input length. Problems for which a polynomial bound doesen't exist or isn't known are said to be *hard* for the computational model. For example, finding solutions to boolean equations (the SAT problem) is believed to be hard for deterministic Turing machines, but it is easy for non-deterministic ones. Unfortunately, non-deterministic Turing machines (along with any other non-deterministic model of computation) are more of a mathematical tool than anything, and there seems to be no practical way to efficiently solve all of the problems which would take Non-deterministic Polynomial time (NP-complete problems). This is well expressed by the strong Church-Turing thesis: no *physical* computational model can be exponentially faster than deterministic Turing machines. Of course, finding a model which is able to solve NP-complete problems would in fact mean that we have found a way to implement non-determinism, which is largely belived to be impossible. On the other hand, there are problems which lie in a 'gray zone' between NP-COMPLETE and P: they are believed to be hard for deterministic models (hence, they are not in P), but there seems to be no way to show that they are NP-COMPLETE. The most famous of such problems is factorization of natural numbers: Shor's algorithm is an hybrid quantum algorithm that can factorize numbers in polynomial time. While still far from usable in practical cases, its existence proves that one must be extremely careful when talking about the hardness of problems, especially when applied to cryptography.

2.2 Hash functions

Hash functions are a fundamental tool in many fields of computer science, and cryptography is arguably the most prominent. Formally, an hash function is any function $H\colon \{0,1\}^* \mapsto \{0,1\}^n$, that is any function which maps arbitrarly long *messages* to fixed-size *digests*. From the definition, it is immediate to see that there are an infinite number of messages which map to the same digest. While an operation like truncation is a (very simple) hash function, in cryptography we are interested in functions that provide additional guarantees: the assumption is that a digest sohuld represent a message in a one-way fashion:

while there are infinite messages which map to the same digest, it must be hard to find them. Ideally, a cryptographic hash function should behave like a perfect random function. This is of course impossible, as the output of an hash function must only depend deterministically on its input; the aim then is to build functions which are hard to distinguish from a random function.

Definition 1 (Cryptographic hash function)**.** Given $n \in \mathbb{N}$, an n(-bit) cryptographic hash function (CHF) is any function $H \colon \{0,1\}^* \mapsto \{0,1\}^n$ which satisfies the following properties:

- **Collision resistance**: It is hard to find two messages m_1, m_2 such that $H(m_1) = H(m_2)$.
- **Preimage resistance**: Given some digest h, it is hard to find a message m such that $H(m) = h$ (H is a one-way function).
- **Second preimage resistance**: Given some message m_1, it is hard to find a message m_2 such that $H(m_1) = H(m_2)$.

While some of the requirements might seem redundant (for example, if it is hard for an attacker to find a collision for chosen messages, it must be hard when one is fixed), the difference usually lies in how exactly we define hardness for each property. For collision resistance, an ideal CHF requires about $2^{n/2}$ evaluations to find a collision (birthday paradox), while for preimage resistance it would require about 2^n evaluations. Tipically, a CHF is built by applying some known secure constructions to functions which are simpler to devise.

Definition 2 (Pseudorandom keyed permutation)**.** Given $l, n \in \mathbb{N}$, an l/n(-bit) pseudorandom keyed permutation (PKP) is any bijective function:

$$F \colon \{0,1\}^l \times \{0,1\}^n \mapsto \{0,1\}^l$$

which is hard to distinguish from an uniform random distribution.

PKPs are often built by iterating a keyed permutation F for some number r of rounds, since F by itself might be relatively easy to invert. A block cipher is a pseudorandom keyed permutation which changes the key being used in each round through a key-scheduling function. Unkeyed permutations can be derived from keyed ones simply by fixing the key to some arbitrary value.

Definition 3 (One-way compression function)**.** Given $l_1, l_2, n \in \mathbb{N}$, an $l_1/n/l_2$(-bit) one-way compression function (OWCF) is any function:

$$F \colon \{0,1\}^{l_1} \times \{0,1\}^n \mapsto \{0,1\}^{l_2}$$

There are many known ways to build OWCFs from pseudorandom keyed permutations, and, in turn, CHFs from OWCFs. We are going to use the Davies-Meyer and the Merkle-Damgård constructions respectively.

Theorem 1 (Davies-Meyer construction)**.** *Given a l/n pseudorandom keyed permutation E, some $i, k \in \mathbb{N}$, some $v \in \{0,1\}^l$, and some $m \in \{0,1\}^{kn}$, then any function F_E such that:*

$$F_{E,i}(v,m) = \begin{cases} E(v, m_1) & i = 1 \\ E(F_{E,i-1}(v,m), m_i) & 2 \leq i \leq k \end{cases}$$
$$F_E = F_{E,k}$$

is a $l/kn/l$ OWCF.

Theorem 2 (Merkle-Damgård construction)**.** *Given a $l_1/n/l_2$ OWCF F, some $v \in (0,1)^{l_1}$, some $m \in (0,1)^*$ and some padding function:*

$$P(m) \colon (0,1)^{|m|} \mapsto (0,1)^{|m|+(|m| \bmod l_1)}$$

such that:

$$\forall m, m' \in \{0,1\}^* \colon (|m| = |m'| \Rightarrow |P(m)| = |P(m')|) \wedge (|m| \neq |m'| \Rightarrow m_{|P(m)|} \neq m'_{|P(m')|})$$

then any function H_F such that:

$$H_{F,i}(v,m) = \begin{cases} F(v, m_1) & i = 1 \\ F(H_{F,i-1}(v,m), m_i) & 1 < i \leq |P(m)| \end{cases}$$
$$H_F = H_{F,|P(m)|}$$

is a CHF.

For any message m, the digest $h = H(m)$, where H is an n CHF, can be used as a *binding commitment* to m in a prover-verifier scenario: the verifier is convinced, with overwhelming confidence $(1 - \frac{1}{2^n})$ that the prover knows m simply by asking him to share h. What if the prover wanted to commit to a list of k messages? Of course, he could share with the verifier an hash for each one of the messages: this would require an $\mathcal{O}(k)$ communication cost and a $\mathcal{O}(k)$ verification cost. A better alternative would be for the prover to share $H(\{m_1, \ldots, m_k\})$: in this case we would have a $\mathcal{O}(1)$ communication cost, but still have a $\mathcal{O}(k)$ verification cost. Furthermore, the verification step cannot be parallelized. For this reason, tree hash modes have been introduced to both reduce verification cost and allow for parallel computation.

Definition 4 (Merkle Tree)**.** Given $k \in \mathbb{N}$, a CHF H and a set of messages $S = \{m_1, \ldots, m_{|S|}\}$, with $|S| = 2^{k-1}$, a Merkle Tree (MT) is a complete binary tree of height k such that:

1. The leaf nodes $\nu_1, \ldots, \nu_{2^{k-1}}$ contain $H(m_1), \ldots, H(m_{2^{k-1}})$.
2. Every other node ν contains $H(\nu_l, \nu_r)$, where ν_l is the left child of ν and ν_r is the right child of ν.

By using Merkle trees, the prover only needs to send to the verifier, as a commitment for some message m_i among $k = 2^{\lfloor \log_2(k) \rfloor}$ messages, the contents of the co-path from the leaf containing m_i to the root (plus the hash of m_i): this requires just $\mathcal{O}(\log_2(k))$ communication effort and $\mathcal{O}(\log_2(k))$ verification effort.

Definition 5 (Augmented Binary tRee)**.** Given $k \in \mathbb{N}$, a CHF H, and a set of messages $S = \{m_1, \ldots, m_{|S|}\}$, with $|S| = 2^{k-1} + 2^{k-2} - 1$, an Augmented Binary tRee (ABR) is a complete binary tree of height k augmented with *middle* nodes, such that:

1. The leaf nodes $\nu_1, \ldots, \nu_{2^{k-1}}$ contain $H(m_1), \ldots, H(m_{2^{k-1}})$.
2. There are no middle nodes in the leaf layer.
3. The middle nodes $\nu_{2^{k-1}+1}, \ldots, \nu_{|S|}$ contain $H(m_{2^{k-1}+1}), \ldots, H(m_{|S|})$.
4. Every other node ν contains $H(\nu_l \oplus \nu_m, \nu_r \oplus \nu_m) \oplus \nu_r$, where ν_l is the left child of ν, ν_r is the right child of ν, and ν_m is the middle child of ν, or 0 if ν doesen't have a middle child.

ABRs can store 50% more messages than Merkle Trees for the same height, resulting in the same number of calls to H, at the cost of performing 3 additional $\oplus$ operations for every call (we assume that $TIME(\oplus) \ll TIME(H)$).

3 ZK-SNARK systems

We saw in Section 2 how a prover can convince a verifier about the knowledge of some message m, with a high confidence and a small communication effort, by using a CHF H. However, the underlying assumption was that m is known by the verifier: when the prover sends h, the verifier can check whether $H(m) = h$ and therefore accept or reject.

Definition 6 (Zero-Knowledge Proof)**.** Given two parties, called the prover P and the verifier V, a secret x, known only to P, and some statement σ of whose truth P wants to convince V by means of some proof π, we call a Zero-Knowledge Proof (ZKP) system any protocol which satisfies the following properties:

- **Soundness**: $\neg\sigma \implies V(\pi) = \bot$.
- **Completeness**: $\sigma \implies V(\pi) = \top$.
- **Zero-Knowledge**: It is *hard* for V to derive x given σ and π.

While formal proofs have been known for millenia, only in the last century, with the advent of modern cryptography, researchers started considering the possibility of having proofs of statements which, while able to convice someone of their truth, didn't leak information about how they were obtained. Zero-Knowledge systems proves particularly useful in *ARgument of Knowledge* (ARK) scenarios (together, they are called ZK-ARK): the prover P wants to convince the verifier V that he knows a solution to some problem, assuming there is one, without revealing the solution itself. For example, P might want to convince V that he knows an assignment of x which satisfies the equation:

$$x^2 - 3x + 2 = 0$$

without revealing the assignment. Of course, in this example it would be easy for V to find the solutions $\{1, 2\}$, reconstruct the proof, and finally discern which of the two solutions was known by P. We assume some familiarity with Turing machines, the Turing thesis and the P $\overset{?}{=}$ NP question, but we'll quickly recall the parts which are most important for us: an NP-COMPLETE problem is a problem for which it is (thought to be) hard to find solutions but it is easy to verify that an alleged solution is in fact one. A ZK-ARK system would allow P to convince V that he knows a solution to an instance of some NP-COMPLETE problem, without giving it away. For example, if P wants to prove that he knows some value of x which satisfies the equation:

$$F(x) = 0$$

where F is a OWF, it would be too hard for V to do what we discussed in the previous example to retrieve the value of x known by P. It must be noted though that known ZK-ARK systems though do not guarantee the formal soundness of the proof: there is a small probability that, given some false statement σ

and an (invalid) proof π, then $V(\pi) = \top$, but this probability is usually in the order of 2^{-128} or even less. There are other nice additional properties that zero-knowledge systems might satisfy, making them even more interesting.

Definition 7 (ZK-SNARK)**.** Given a prover P, a verifier V, a statement σ, and a ZK-ARK system to produce a proof π, if the system is:

- **Succint**: $SPACE(\pi) = o(\log(\sigma))$.
- **Non-interactive**: The only communication required by the system is the exchange of σ and π.

then it is a Zero-Knowledge Non-interactive ARgument of Knowledge (ZK-SNARK) system.

Succintness is an important property in a blockchain scenario, since we cannot afford to use too much space to store the proofs, and non-interactivity of the process allows for efficient verification when multiple parties are involved.

One of the most important applications of ZK-SNARK systems is in *provable computation*, where the prover wants to convince the verifier that he correctly performed some computation (e.g. a cryptocurrency transaction). A very famous ZK-SNARK system for verifiable computation is the *Pinocchio* protocol, which was the first one efficient enough to be practical.

3.1 The Pinocchio Protocol

Pinocchio is composed of many different components, and requires quite a bit of mathematical background to be fully understood. We will not go into all of the mathematical and cryptographic details of the protocol, especially the ones involving *elliptic curves*, but we will still try to give a good idea of how the protocol works, and ultimately what determines its computational complexity.

Definition 8 (Prime field)**.** Given a prime number p, the associated prime field is the set $\mathbb{F}_p = \{\{0, \dots, p-1\}, \oplus, \otimes\}$, where $\oplus$ is integer addition modulo p and $\otimes$ is integer multiplication modulo p.

For ease of notation, we will often use $+$ in place of $\oplus$ and omit $\otimes$ if it is clear from the context.

Definition 9 (Arithmetic circuit)**.** Given a field $\mathbb{F}$, some $n, m \in \mathbb{N}$, some constants $a_{1,1}, \dots, a_{m,n} \in \mathbb{F}$, and some variables $x_1, \dots, x_n$ over $\mathbb{F}$, an arithmetic circuit over $\mathbb{F}$ is any formula ϕ of the type:

$$\begin{aligned}
&\phi \equiv c && \text{with } c \in \mathbb{F} \\
&\phi \equiv x && \text{with } x \text{ variable over } \mathbb{F} \\
&\phi \equiv \phi' \oplus \phi'' && \text{with } \phi' \text{ and } \phi'' \text{ arithmetic circuits} \\
&\phi \equiv \phi' \otimes \phi'' && \text{with } \phi' \text{ and } \phi'' \text{ arithmetic circuits}
\end{aligned}$$

Every arithmetic circuit can be represented by a Directed Acyclic Graph (DAG), where the vertices are labeled either with a variable, a constant or one of the operations $\oplus$ and $\otimes$: in the latter case, the vertex must have exactly two incoming edges which come from the vertices representing the inputs of the operation.

Pinocchio does not allow the encoding of arbitrary languages, i.e. it is not Turing complete, but we are restricted to arithmetic circuits over an arbitrary prime field $\mathbb{F}_p$. The main limitation arising from this restriction is that we cannot express unbounded computation (e.g. loops whose exit condition depends on some non-constant value) in this framework. This issue can be mitigated by writing a circuit compiler in a Turing complete language which is able to synthesize parametrized arithmetic circuits on the fly.

Definition 10 (Rank-1 Contraint System)**.** Given a field $\mathbb{F}$ and some $m, n \in \mathbb{N}$, any set:

$$\{(a_1, b_1, c_1), \dots, (a_n, b_n, c_n) \mid \forall i\colon a_i, b_i, c_i \in \mathbb{F}^m\}$$

is an n/m Rank-1 Constraint System (R1CS) over $\mathbb{F}$. Given an R1CS $\mathcal{C}$, a *solution* to $\mathcal{C}$ is any vector:

$$s \mid s \in \mathbb{F}^m \wedge \forall i\colon (s \cdot a_i)(s \cdot b_i) = s \cdot c_i$$

where $\cdot$ is the dot product operation.

Fundamentally, an R1CS is a system of linear equations. Any arithmetic circuit with n multiplicative gates and $m - 1$ variables can be associated with an n/m R1CS (the extra variable is the constant 1 of the chosen field) in the following way:

1. Multiplications by constants are unrolled into chains of additions.
2. Chains of addition nodes are collapsed at multiplicative nodes.
3. $\forall i, j\colon a_{i,j}$ will contain the coefficient with which the jth variable is input to the *left* of the ith multiplicative gate.
4. $\forall i, j\colon b_{i,j}$ will contain the coefficient with which the jth variable is input to the *right* ith multiplicative gate.
5. $\forall i, j\colon c_{i,j}$ will contain the coefficient with which the jth variable is output from the ith multiplicative gate.

Let's make an example to better understand the process.

Example 1. *We have the prime field $\mathbb{F}_{13}$ and want to compute the function:*

$$f(x_1, x_2) = x_2(x_1^3 + 4x_2 + 5)$$

The corresponding arithmetic circuit is:

$$x_2(x_1 x_1 x_1 + 4x_2 + 5)$$

Note that $4x_2 = x_2 + x_2 + x_2 + x_2$, so multiplications by constants don't really count as multiplications! Let's explicit all the intermediate variables (i.e. the outputs of the multiplications):

$$t_1 = x_1 x_1 \qquad t_2 = t_1 x_1 + 4x_2 + 5 \qquad y = t_2 x_2$$

We can see that there is a total of 3 multiplication gates and 5 variables (two input, two intermediates, one output), plus the implicit variable representing the constant 1*. We can now build the associated* $3/6$ *R1CS. The first constraint of the system is:*

$$a_1 = (0,1,0,0,0,0) \qquad b_1 = (0,1,0,0,0,0) \qquad c_1 = (0,0,0,1,0,0)$$

Since we are multiplying x_1 (represented by the second element in the vectors) by itself and putting it into t_1, which is the fourth element. Similarly, we build the remaining constraints:

$$\begin{aligned} a_2 &= (0,0,0,1,0,0) \qquad & b_2 &= (0,1,0,0,0,0) \qquad & c_2 &= (8,0,9,0,1,0) \\ a_3 &= (0,0,0,0,1,0) \qquad & b_3 &= (0,0,1,0,0,0) \qquad & c_3 &= (0,0,0,0,0,1) \end{aligned}$$

(a_2, b_2, c_2) *might confuse at first, but it is easy to derive once we see that:*

$$t_2 = t_1 x_1 + 4x_2 + 5 \iff t_2 - 4x_2 - 5 = t_1 x_1 \iff 8 + 9x_2 + t_2 = t_1 x_1$$

Remember that we are working over $\mathbb{F}_{13}$: $-4 \equiv 9$ and $-5 \equiv 8$. With this, we have successfully built our target R1CS.

Suppose now that we want to prove that we know x_1, x_2 such that $y = f(x_1, x_2) = 10$. For example, $x_1 = 2, x_2 = 3$ are valid choices, since:

$$3(2^3 + 4 \times 3 + 5) = 75 \equiv 10 \pmod{13}$$

After computing the intermediates values $t_1 = 4$ and $t_2 = 25 \equiv 12 \pmod{13}$, we can find the solution:

$$s = (1,2,3,4,12,10)$$

The reason we translate arithmetic circuits into R1CS is that they explicit all of the computation in terms of linear combinations, which allows us to use Lagrange interpolation to build polynomials over them.

Definition 11 (Polynomial fields)**.** Given a field $\mathbb{F}$, some $n \in \mathbb{N}$ and some $d_1, \dots, d_n \in \mathbb{N}$, we denote with $\mathbb{F}[x_1^{d_1}, \dots, x_n^{d_n}]$ the set of all n-variate polynomials in $\mathbb{F}$ over variables $x_1, \dots, x_n$ each with maximum degree $d_1, \dots, d_n$.

Definition 12 (Quadratic Arithmetic Program)**.** Given a field $\mathbb{F}$ and some $n, m \in \mathbb{N}$, any set:

$$\{t, \{v_1, \dots, v_n\}, \{w_1, \dots, w_n\}, \{y_1, \dots, y_n\}\} \mid t \in \mathbb{F}[x^m] \wedge \forall i\colon v_i, w_i, y_i \in \mathbb{F}[x^{m-1}]$$

is an Quadratic Arithmetic Program (QAP) over $\mathbb{F}$.

With all this in mind, the flow of the Pinocchio protocol is as follows:

1. Encode an algorithm as an arithmetic circuit over some prime field $\mathbb{F}_p$.
2. Compute the associated R1CS.
3. Compute the associated QAP.
4. Generate random values and instantiate an homomorphic encryption mapping (using elliptic curves) which depends on those values.
5. Generate the proof by encrypting the QAP with the homomorphic mapping.
6. Map again the proof to a new homomorphic space, and finally verify it.

Due to the homomorphism of the mappings and the properties of QAPs and R1CSs, if the verification is successful, it means that the original algorithm was in fact correctly executed, with high probability. If the verification fails, then certainly the original algorithm was not executed correctly. The verifier learns cannot learn any additional information from this process without performing an infeasible amount of work, therefore this is indeed a ZK-SNARK protocol.

4 The libsnark library

5 Experiments

6 Conclusions and future directions