Formal Verification of Pohlig-Hellman Algorithm for Computing Discrete Logarithms with Coq

Jeremiah Daniel A. Regalario*[1], Marc Emanuel N. Dela Paz[2], Meluisa D. Montealto[3], Nicole Coleen V. Santos[4], and Alfonso B. Labao[5]

[1] Numerical Analysis and Scientific Computing Research Group, Institute of Mathematics, College of Science, University of the Philippines Diliman, Quezon City 1101, Philippines,
jaregalario@up.edu.ph,
WWW home page: https://math.upd.edu.ph
[2] Logic and Computability Laboratory, Department of Computer Science, College of Engineering, University of the Philippines Diliman, Quezon City 1101, Philippines,
mndelapaz@up.edu.ph,
WWW home page: https://dcs.upd.edu.ph
[3] Institute of Mathematics, College of Science, University of the Philippines Diliman, Quezon City 1101, Philippines,
nvsantos@up.edu.ph,
WWW home page: https://math.upd.edu.ph
[4] Logic and Computability Laboratory, Institute of Mathematics, College of Science, University of the Philippines Diliman, Quezon City 1101, Philippines,
nvsantos@up.edu.ph,
WWW home page: https://math.upd.edu.ph
[5] Department of Computer Science, College of Engineering, University of the Philippines Diliman, Quezon City 1101, Philippines,
ablabao@up.edu.ph,
WWW home page: https://dcs.upd.edu.ph

Abstract. With the discrete logarithm problem (DLP) underpinning many modern cryptographic protocols, ensuring the correctness of algorithms that solve DLP is crucial for digital security. In this work, we present a machine-checked formalization of the *Pohlig–Hellman algorithm* in the Coq proof assistant. We introduce an abstract FiniteCyclic Group class to model cyclic groups of known order $\mathbb{Z}\not{p}$, then invoke key number-theoretic results (*Fundamental Theorem of Arithmetic*, *Fermat's Little Theorem*, and the *Chinese Remainder Theorem*) as axioms to reconstruct the Pohlig-Hellman index factorization approach. Our proof shows that, given a generator and any group element, Coq's CRT solver and a discrete black box log oracle in prime power subgroups combine to recover the unique exponent satisfying $g^x = h$. This development not only demonstrates how Coq can faithfully capture nontrivial cryptographic reasoning.

Keywords: discrete logarithm, Pohlig–Hellman algorithm, Coq formalization, cyclic groups, Chinese Remainder Theorem, cryptography

J. Caro et al. (eds.), *Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2025)*, Atlantis Highlights in Computer Sciences 24,
https://doi.org/10.2991/978-94-6239-638-8_8

1 Introduction

The *discrete logarithm problem (DLP)* is one of the main foundations of modern cryptography, and is the idea behind the security of protocols such as Diffie-Hellman key exchange, ElGamal encryption, and elliptic curve cryptography. Its computational difficulty ensures the confidentiality and integrity of digital communications, enabling applications like secure messaging, digital signatures, and privacy-preserving technologies such as zero-knowledge proofs.

The *Pohlig-Hellman algorithm*, which efficiently solves DLP in groups with composite-order subgroups, plays a crucial role: it highlights vulnerabilities in improperly configured cryptographic systems while guiding the design of robust, attack-resistant primitives. This algorithm was first introduced by Roland Silver, but was later published in 1978 by Stephen Pohlig and Martin Hellman [1]. Formally verifying this algorithm is critical to ensure its correctness and reliability, as any flaw in its implementation could lead to errors such as misassessments of cryptographic security. By employing rigorous mathematical methods with Coq to confirm the algorithm's adherence to theoretical foundations, formal verification safeguards trust in cryptographic standards, ensuring that security evaluations accurately reflect real-world risks [2].

1.1 The Coq Proof Assistant

This paper presents a formal verification of the Pohlig-Hellman algorithm using the Coq proof assistant. The Coq proof assistant is a tool that is used to formally verify mathematical theorems using logical statements in its rigorous proof framework. Coq is an interactive proof assistant based on the *Calculus of Inductive Constructions (CIC)*: a dependently-typed functional language (Gallina) together with a certified proof-checker. In Coq, you can both "define" mathematical objects and algorithms as executable Gallina code and "state and machine-check" formal propositions about them. Proofs are written as scripts that build a proof term which the kernel verifies down to the smallest inference rule. Formal verification in Coq ensures that correctness of the theorem in all cases of possible inputs along with its execution, which gives high assurance that every correctness claim about the Pohlig–Hellman algorithm is mechanically checked, eliminating possible inevitable gaps in informal proofs and catching subtle corner cases that are easy to miss on paper. Cryptography relies heavily on mathematical guarantees [3]. Hence, small errors in the logic or in its proof invalidate the security guaranteed by the cryptographic method. Using Coq to verify cryptographic algorithms such as the Pohlig-Hellman ensures that both the mathematical properties and algorithmic behaviors are consistent with their formal specifications. In addition, Coq's rich type system (and supporting libraries such as MathComp) makes it convenient to formalize the algebraic structures (finite cyclic groups, orders, and modular arithmetic) on which Pohlig-Hellman depends, and its proof automation and extraction features allow us both to shorten routine reasoning and produce verified executable code from formalization. [4]

1.2 Significance of the Study

The significance lies in its demonstration of the limitations and strengths of the algorithm, particularly the importance of using subgroups of large prime order to ensure cryptographic security. By formally verifying Pohlig-Hellman, the study underscores how certain systems become vulnerable when small prime factors are present in the group order. Moreover, it contributes to the broader effort of applying formal methods in cryptography, offering a concrete example of how tools like Coq can be used to rigorously verify nontrivial algorithms. In doing so, this work connects theoretical number theory with practical concerns in cryptographic design and analysis [5]

Several efforts have been made recently that used interactive proof assistants to give machine-checked guarantees for cryptographic constructions and their implementations. Notably, CertiCrypt and related Coq frameworks enable fully formal, game-based security proofs inside Coq, providing a foundation for mechanized proofs of probabilistic, code-based arguments. The Foundational Cryptography Framework (FCF) similarly embeds probabilistic programs and game-playing proof techniques in Coq to produce concrete, computational security bounds [6]. On the implementation side, projects such as Fiat Cryptography / Fiat-Crypto use Coq to synthesize and verify high-performance finite-field and modular-arithmetic code used in real-world cryptographic libraries, while other toolchains (e.g., EasyCrypt [21] and the Everest/HACL*/EverCrypt [22] stack) provide complementary approaches for reasoning about security or for producing verified, efficient implementations (some using other proof-oriented languages such as F*). Work verifying concrete primitives and implementations (e.g., verified SHA-256/AES [24] developments and verified C code using separation-logics) shows the feasibility and practical value of combining formal proofs with verified implementations. Our Coq formalization of the Pohlig–Hellman algorithm complements these lines of work by providing a machine-checked correctness and algebraic analysis of the discrete-log reduction strategy itself; unlike frameworks that target full security proofs for protocol/KEM families or verified low-level implementations, our contribution focuses on the algorithmic correctness and algebraic lemmas needed specifically for Pohlig–Hellman, and can be reused by future Coq-based security proofs or verified implementations that rely on discrete-log subroutines.

1.3 The Discrete Logarithm Problem (DLP)

The DLP statement is as simple as finding an integer a such that

$$\alpha^a \equiv \beta \pmod{m}, \tag{1}$$

where we define $a := \log_\alpha \beta$. This statement can be relaxed by setting $m = p$, where p is prime and assuming that α is a primitive root $\bmod m$ and restricting $a \in \mathbb{N}$. [9]

2 The Pohlig-Hellman Algorithm to solve DLP

2.1 Preliminaries

In this paper, we denote $\mathbb{N} = \{0, 1, 2, \dots\}$.

Definition 1 (Finite Cyclic Group).

1. *A binary operation* $*$ *on a set* A *is a function* $* : A \times A \to A$.
2. *A group is a pair* $\langle G, * \rangle$ *where* $G \neq \emptyset$ *is a set together with a binary operation* $*$ *on* G *such that*
   - G_1: *(Associativity) For all* $a, b, c \in G$, $(a * b) * c = a * (b * c)$.
   - G_2: *(Identity) There exists a unique element* $e \in G$ *such that* $e * g = g * e$ *for any* $g \in G$, *which we call the identity.*
   - G_3: *(Inverse) For any* $g \in G$, *there exists* $h \in G$ *such that* $gh = e = hg$ *and we denote* $g^{-1} := h$ *as the inverse of* g.

   For the next items, let $\langle G, * \rangle$ *be a group.*

3. *If* $H \subseteq G$ *and* $\langle H, * \rangle$ *also forms a group, then* H *is said to be a subgroup of* G, *denoted by* $H \leq G$
4. *Let* $n \in \mathbb{Z}$ *and* $g \in G$. *Then,*

$$g^n = \begin{cases} \underbrace{g * g * \cdots * g}_{n \text{ times}}, & \text{if } n > 0 \\ 1, & \text{if } n = 0 \\ \underbrace{g^{-1} * g^{-1} * \cdots * g^{-1}}_{|n| \text{ times}}, & \text{if } n < 0 \end{cases}$$

5. *For an element* $g \in G$, *a cyclic subgroup generated by* g *is denoted by*

$$\langle g \rangle := \{g^k : k \in \mathbb{Z}\}.$$

6. *A group* G *is said to be cyclic if there exists* $g \in G$ *such that* $G = \langle g \rangle$.
7. *The order of a group* $\langle G, * \rangle$ *is given by the cardinalitty of the set* G, *i.e.* $|G|$. *A group* G *is said to be of finite order if* $|G| \in \mathbb{N}$.
8. *The order of an element* $g \in G$ *is the smallest positive integer* n *such that* $g^n = e$. *We denote the order as* $|g| := n$.
8. *Any finite cyclic group of order* n *is isomorphic to the additive group* $\langle \mathbb{Z}_n, +_n \rangle$ *where* $\mathbb{Z}_n := \{0, 1, \dots, n\}$ *and* $+_n$ *is the addition modulo* n.
9. *A special class of finite of finite cyclic group is a multiplicative group. If* n *is a power of a prime, then the multiplicative group given by* $\langle \mathbb{Z}_{p^k}^{\times}, \cdot_{p^k} \rangle$ *where* $\mathbb{Z}_{p^k}^{\times} = \{1, 2, \dots, p^k\}$ *and* $\cdot_{p^k}$ *is the multiplication modulo* p^k *is a finite cyclic group of order* $p^{k-1}(p-1)$.

Definition 2. *A primitive root modulo* n *is an element* $p \in \mathbb{Z}_n^{\times}$ *s.t.* $|p| = |G|$.

Definition 3. *Let $\langle G, * \rangle$ and $\langle G', *' \rangle$ be groups. A function $\varphi : \langle G, * \rangle \to \langle H, * \rangle$ is an isomorphism of groups if $\varphi : G \to H$ is a bijection and for any $g_1, g_2 \in G$, we have*

$$\varphi(g_1 * g_2) = \varphi(g_1) *' \varphi(g_2).$$

We say that G is isomorphic to H if there exists such isomorphism, i.e. $G \cong H$.

Theorem 1 (Fundamental Theorem of Arithmetic). *For every $n \in \mathbb{N}$ such that $n > 1$, there exists primes $p_1, p_2, \ldots, p_k$ such that*

$$n = p_1^{c_1} p_2^{c_2} \cdots p_k^{c_k}$$

for some $k \in \mathbb{N}$, where $c_1, c_2, \cdots, c_k \in \mathbb{N}$ are the corresponding prime exponents. This factorization is unique, except possibly by the order in which the factors appear and with the multiples of units (± 1).

Theorem 2 (Chinese Remainder Theorem). *Let $m_1, \ldots, m_n > 1$ be pairwise coprime positive integers such that $x, a_1, a_2, \ldots, a_n \in \mathbb{Z}$ satisfy the following system of congruences:*

$$\begin{cases} x & \equiv a_1 \pmod{m_1} \\ x & \equiv a_2 \pmod{m_2} \\ & \vdots \\ x & \equiv a_n \pmod{m_n}. \end{cases}$$

The system above has a unique solution under modulo $M = m_1 m_2 \cdots m_n$.

Equivalently, with the mapping $x \bmod M \mapsto (x \bmod m_1, , \ldots, x \bmod m_n)$:

$$\mathbb{Z}_M \cong \mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_n}.$$

Theorem 3 (Fermat's Little Theorem). *Let p be prime. For any $n \in \mathbb{Z}_p^\times$, we have*

$$n^{|\mathbb{Z}_p^\times|} = 1.$$

Alternatively, for any $n \in \mathbb{Z}$ relatively prime to p, we have

$$n^{p-1} \equiv 1 \pmod{p}.$$

2.2 The Pohlig-Hellman Algorithm

Let p be a prime and α be a primitive root $\bmod p$. By Fundamental Theorem of Arithmetic (FTOA), we can factor $p - 1$ as:

$$p - 1 = p_1^{c_1} p_2^{c_2} \cdots p_k^{c_k}, \tag{2}$$

where p_i's are distinct primes.

The main idea of the Pohlig-Hellman algorithm is to reduce the original DLP modulo p into smaller DLPs modulo $p_i^{c_i}$ and then reconstruct the solution

modulo $p-1$ using the Chinese Remainder Theorem (CRT). In other words, find $a \bmod p_i^{c_i}$ for each $i \in \{1, \ldots, k\}$, then we use the CRT to combine them together, and the result will follow from Fermat's Little Theorem (FLT):

$$a^{p-1} \equiv 1 \pmod{p} \tag{3}$$

assuming the $\gcd(a, p) = 1$.

From here, we can reduce the algorithm by focusing on a particular substep, i.e. for $q = p_i$ and $c = c_i$.

We wish to find $x = a \bmod q^c$.

1. Express $x = \sum_{i=0}^{c-1} a_i q^i$ so that
$$a = a_0 + a_1 q + a_2 q^2 + \cdots + a_{c-1} q^{c-1} + s q^c, \tag{4}$$
for some integer s (here, s is arbitrary and supports the non-uniqueness of discrete logarithms)
2. Compute for a_0 by solving
$$\beta^{\frac{p-1}{q}} \equiv \alpha^{\frac{a_0(p-1)}{q}} \pmod{p}. \tag{5}$$
3. Now, we define $\beta_0 = \beta$. To compute for $beta_j$ where $j \in \{0, \ldots, c-1\}$,
$$\beta_j = \beta \alpha^{-(a_0 + a_1 q + \cdots + a_{j-1} q^{j-1}))} \pmod{p} \tag{6}$$
Using this, we find a_j using the congruence
$$\beta_j^{\frac{p-1}{q^{j+1}}} \equiv \alpha^{\frac{a_j(p-1)}{q}} \pmod{p} \tag{7}$$
Now, given with β_j and a_j, we can find β_{j+1} with the recurrence relation given by
$$\beta_{j+1} = \beta_j \alpha^{-a_j q^j} \pmod{p}. \tag{8}$$

After computing all $a_0, a_1, \ldots, a_{c-1}$, we reconstruct: $x = \sum_{i=0}^{c-1} a_i q^i$. Repeat the process for all prime powers $p_i^{c_i}$ dividing $p-1$, and finally recover a by solving the system:

$$a \equiv x_i \pmod{p_i^{c_i}} \tag{9}$$

for all i using CRT.

3 Formal Verification

3.1 Preliminaries

First, we lay out the necessary built-in modules from Coq. These are essential for handling lists, natural number arithmetic, and Euclidean properties.

```COQ
Require Import Coq.Lists.List.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.PeanoNat.
Import ListNotations.
```

- `Coq.Lists.List` provide list-related types and functions.
  - ▶ This brings in Coq's standard library for lists: the type `list A`, the constructors `nil` and `cons`, and operations like `nth_error`, `length`, `map`, etc.
  - ▶ Later we'll use `nth_error`, `map`, and `Forall2` on lists of prime-power pairs.
- `Coq.Arith.Arith` includes basic arithmetic over natural numbers.
  - ▶ Exports basics of arithmetic on nat (Peano naturals), including addition, multiplication, exponentiation (`Nat.pow`), and comparison functions.
- `Coq.Arith.PeanoNat` gives additional results on `nat` (natural numbers).
  - ▶ Provides supplementary arithmetic facts (e.g. `Nat.div`, `Nat.modulo`, `Nat.gcd`).
  - ▶ We need `Nat.gcd` when showing that different prime-power factors are pairwise coprime.
- `ListNotations`: Introduces the "`[x; y; z]`" and "`x :: xs`" notations for lists.

For the next parts of our code, we will enclose everything in `PohligHellman` section as follows while invoking the context of a `FiniteCycliCGroup.`

```COQ
Section PohligHellman.
  Context `{FiniteCyclicGroup}.
  ...

End PohligHellman.
```

In addition, we will also formally define the concept of primes and primitive roots.

```COQ
(* Primality *)
Inductive prime : nat -> Prop :=
| prime_2 : prime 2
| prime_gt_2 : forall n,
    2 < n ->
    (forall p q, n = p * q -> p = 1 \/ q = 1) ->
    prime n.
```

This inductive predicate defines what it means for a natural number to be prime:

- 2 is explicitly declared prime.

- Any number greater than 2 is prime if it cannot be written as a product of two natural numbers other than 1 and itself.

Next, we have primitive roots modulo p.

```
(* Primitive roots modulo p *)
Definition primitive_root (p g : nat) : Prop :=
  prime p ->
  forall a (Ha1 : 1 <= a) (Ha2 : a < p) (Hg : Nat.gcd a
      p = 1),
    exists k : nat, k < p /\ a = Nat.modulo (pow g k) p.
```

This defines the property of a number g being a primitive root modulo p:

- For every integer a relatively prime to p, there exists a power k such that $g^k \equiv amodp$, meaning g generates the multiplicative group modulo p.

Defined Parameters

```
  Parameter discrete_log_primepower : G -> G -> nat ->
      nat -> nat.
```

3.2 Objects

Now, we formally define a structure `FiniteCyclicGroup`, which essentially models the group $\mathbb{Z}_p^\times$, the multiplicative group modulo a prime p, and the main object that we will utilize for modular arithmetic.

```
Class FiniteCyclicGroup := {
  G : Type;
  op : G -> G -> G;
  e : G; (* group identity *)
  inv : G -> G; (* group inverse *)
  pow : G -> nat -> G; (* exponentiation *)

  (* Group axioms: associativity, identity, inverses *)
  assoc : forall x y z : G, op x (op y z) = op (op x y)
      z;
  id_l : forall x : G, op e x = x;
  id_r : forall x : G, op x e = x;
  inv_l : forall x : G, op (inv x) x = e;

  (* Exponentiation axioms *)
  pow0 : forall x : G, pow x 0 = e;
  powS : forall x : G, forall n, pow x (S n) = op x (pow
       x n);

```

```
18    (* The group is cyclic of known (finite) order *)
19    order : nat;
20    gen : G;
21    gen_order : pow gen order = e
22  }.
```

COQ

This abstract class defines:

- **A type** `G` representing the elements of the group.
  - ▶ We declare a new abstract type `G` which will stand for "the carrier type of the cyclic group." In a concrete instance, one could set `G := nat` and interpret `op := Nat.mod_mul p`, `e := 1`, etc.
- **A binary operation** `op`, identity `e`, inverse `inv`, and exponentiation `pow`.
  - ▶ Denotes the group operation (e.g. modular multiplication when $G = \mathbb{Z}_p^\times$).
  - ▶ We require `assoc : forall x y z, op x (op y z) = op (op x y) z` enforces associativity.
- **The identity element** `e`.
  - ▶ The identity element of the group. (e.g. $e = 1$ in $\mathbb{Z}_p^\times$).
  - ▶ We add `id_l : forall x, op e x = x` and `id_r : forall x, op x e = x` to state **left and right identity.**
- **The existence of inverse of elements.**
  - ▶ **Group inverse.** For each `x : G`, `inv x` satisfies `op (inv x) x = e.`
  - ▶ We include `inv_l : forall x, op (inv x) x = e.` (A right-inverse lemma could be added similarly, but one direction suffices here.)
- **Rules for exponentiation: zero and successor.**
  - ▶ Denotes exponentiation in the group: `pow g n = g^n.`
  - ▶ **We assume two axioms for exponentiation:**
    - `pow0 : forall x, pow x 0 = e`
    - `powS : forall x n, pow x (S n) = op x (pow x n)`

  These allow inductive reasoning about powers of g.
- **The order:**
  - ▶ Denotes the finite order of the group. In $\mathbb{Z}_p^\times$, order $= p - 1$.
- **The generator gen:**
  - ▶ A chosen generator (primitive root) of the cyclic group.
- `gen_order : pow gen order = e`
  - ▶ the axiomatic invokation of Fermat's Little Theorem
  - ▶ this records that `gen^order = e.`

3.3 Other Theorems

In this paper, we will assume other build up theorems necessary for this formal verification as **Axiom**, like the **Fundamental Theorem of Arithmetic (FTOA)**, **Fermat's Little Theorem (FLT)**, and the **Chinese Remainder Theorem (CRT)**. These theorems have been proven already by several literatures. The reader may peruse [8] and [7] for the proof for FTOA, [10] and [9] for the proof for FLT, [12] and [11] for the proof for CRT.

Discrete log solver for prime-power order subgroup of order p^k

```COQ
Parameter discrete_log_primepower :
  G -> G -> nat -> nat -> nat.

Axiom discrete_log_primepower_correct :
  forall (g h : G) (p k : nat),
  let m := p ^ k in
  pow g m = e ->
  exists x, x < m /\ pow g x = h.

Axiom discrete_log_primepower_spec :
  forall (g h : G) (p k : nat),
  let m := p ^ k in
  pow g m = e ->
  let x := discrete_log_primepower g h p k in
  x < m /\ pow g x = h.
```

- The `discrete_log_primepower g h p k` solves x such that $g^x \equiv h \pmod{p^k}$ where g has order p^k.
- The correctness axiom states that under the assumption $g^{p^k} = e$, such an x exists and is less than p^k.
- Satisfaction of spec axiom essentially tells Coq that the return value `discrete_log_primepower g h p k` satisfies the spec: it is $x < m$ and $g^x = h$.
- The algorithm will rely on this to extract the local exponent x_i at each prime-power factor.

Some notes here:

1. We treat `discrete_log_primepower` as a blackbox solver on subgroups of order p^k, assuming its correctness by axiom.
2. In practice, one could prove `discrete_log_primepower_correct` by reducing to repeated baby-step giant-step calls or using number-theoretic lemmata, but for clarity, in this paper, we keep it as an axiom.

The Chinese Remainder Theorem (CRT)

```COQ
Parameter CRT_solve : list nat -> list nat -> nat.

Axiom CRT_solve_correct:
  forall mods rems,
  length mods = length rems ->
  (* pairwise coprime *)
  (forall i j, i <> j -> Nat.gcd (nth i mods 1) (nth j
      mods 1) = 1) ->
  let M := fold_right Nat.mul 1 mods in
  let x := CRT_solve mods rems in
```

```
    x < M /\
    Forall2 (fun mi ri => x mod mi = ri) mods rems.
```

- The `CRT_solve mods rems` returns the unique solution $x < \prod_i \text{mods}_i$ to the system of congruences:

$$x \equiv \text{rems}_i \pmod{\text{mods}_i}, \quad \forall i$$

  where the mods_i's are pairwise coprime.
- The axiom ensures the solution x is valid and below the product of all moduli.

Some notes here:

1. `CRT_solve_correct` packages the Chinese Remainder Theorem: given a list of pairwise coprime moduli and corresponding remainders, it returns a unique solution below the product of the moduli.
2. We will later set `mods = [p_1^{c_1}; p_2^{c_2}; ...; p_k^{c_k}]` and `rems = [x_1; x_2; ...; x_k]`, where each x_i is the discrete log modulo $p_i^{c_i}$.

3.4 The Main Algorithm

This is the core implementation of the Pohlig–Hellman algorithm using the assumptions and components above.

```
Definition discrete_log (h : G) : nat :=
  let n := order in
  let facs := factor n in
  let mods := map (fun pk => (fst pk) ^ (snd pk)) facs
      in
  let rems := build_rems facs h in
  CRT_solve mods rems.
```

- **order** is the known order n of the group.
- **factor** n returns a list of prime-power decompositions $[(p_1, k_1), \ldots]$, we denote facs:

$$\prod_{(p,c)\in facs} p^c = n.$$

- **mods** is the list of p^k values.
- **rems** computes each x_i such that $g_i^{x_i} = h_i$ in the subgroup of order p^k.
- The final solution is reconstructed using **CRT_solve**.

3.5 Verifier

For verification, first we need the helper function **build_rems**:

```COQ
Fixpoint build_rems (facs : list (nat * nat)) (h : G) :
    list nat :=
  match facs with
  | [] => []
  | (p, k) :: rest =>
      let m := p ^ k in
      let g_i := pow gen (order / m) in
      let h_i := pow h (order / m) in
      discrete_log_primepower g_i h_i p k
      :: build_rems rest h
  end.
```

- The function `build_rems` visits each prime-power factor (p, k).
- It computes $g_i = gen^{n/p^k}$ and $h_i = h^{n/p^k}$, then calls the subroutine `discrete_log_primepower` to extract the exponent x_i.
- These local exponents are collected in a list `rems`, which is later fed to `CRT_solve`.

And now, for the correctness proof, we have the theorem `discrete_log_correct` as:

```COQ
Theorem discrete_log_correct :
  forall (h : G),
  pow gen (discrete_log h) = h.
```

The theorem essentially verifies that the value given by the Pohlig–Hellman algorithm correctly solves the discrete logarithm problem. The proof proceeds as follows: [16]

```COQ
Theorem discrete_log_correct :
  forall (h : G),
  pow gen (discrete_log h) = h.
Proof.
  intros h. (* Introduce the arbitrary element h : G *)

  (* Expand the definition of `discrete_log h` *)
  unfold discrete_log.
  set (n := order). (* Let n = order of the group *)

  (* Let facs be the prime-power factors of n *)
  set (facs := factor n).
  set (mods := map (fun pk => (fst pk)^(snd pk)) facs).
  set (rems := build_rems facs h).
```

At the very start, we let n be the known order of the group, factor it into prime-power components (giving `facs`), then extract the moduli (`mods`) and the local discrete logs (`rems`). The final result is `CRT_solve mods rems.` [19]

1. Use Factor-Oracle to Break Down n

```
(* 1. factor_correct gives Product (p^k) = n. *)
destruct (factor_correct n) as [HprimeH Hprod].
```

factor_correct n returns a pair of facts:

- `Forall (fun pk => prime (fst pk)) facs` = each base p_i in the factor list is prime.
- `fold_right (fun pk acc => acc * (fst pk) ^ (snd pk)) 1 facs = n`, which means the product of all $p_i^{c_i}$ equals n.

We call these two facts:

- `HprimeH` : "All factors are prime."
- `Hprod` : "All factors are powers of prime."

By `factor_correct`, we know each (p_i, c_i) in `facs` is prime-power, and their product recovers n. This is the formal FTOA step. **2. Show each gen^{n/m_i} has order m_i.**

This step shows that $gen^{n/m}$ generates (or at least lies in) the subgroup of order $m = p^k$. Concretely, $(gen^{n/m})^m = gen^n = e$.

```
(* 2. show for each (p,k) in facs, pow (gen^(n/m)) m = e
    . *)
assert (Horder_i : forall p k, In (p, k) facs ->
  let m := p ^ k in
  pow (pow gen (n / m)) m = e).
{
  intros p k Hin.
  pose proof (factor_divides n p k Hin) as [Hmn_nz
      Hmn_div].
  simpl.
  rewrite <- pow_mul.
  rewrite Hmn_div.
  exact gen_order.
}
```

3. Relate each x_i to $gen^{n/m}$.

```coq
(* 3. Each x_i from build_rems satisfies g_i^{x_i} = h_i
    . *)
assert (Hpp : forall i p k,
  nth_error facs i = Some (p, k) ->
  let m := p ^ k in
  let x_i := discrete_log_primepower (pow gen (n / m))
                                     (pow h (n / m))
                                     p k in
  x_i < m /\ pow (pow gen (n / m)) x_i = pow h (n / m)).
{
  intros i p k Hnth.
  apply nth_error_In in Hnth as Hin.
  pose proof (Horder_i p k Hin) as Hord.
  set (m := p ^ k).
  set (g_i := pow gen (n / m)).
  set (h_i := pow h (n / m)).
  pose proof (discrete_log_primepower_spec g_i h_i p k
      Hord)
        as [Hx_lt Hx_eq].
  split; assumption.
}
```

Here, we match each $(p_i, c_i) \in$ facs to an index i. Then, under the condition `nth_error facs i = Some (p_i, c_i)`, we apply the axiom on `discrete_log_primepower` to get the local exponent x_i in the subgroup of order $p_i^{c_i}$.

4. Show Consistency of Length

```coq
(* 4. Length(mods) = Length(rems). *)
assert (Hlen : length mods = length rems).
{
  subst mods rems.
  rewrite map_length, build_rems_length.
  reflexivity.
}
```

This essentially verifies that the length of the moduli $[p_i^{c_i}]$ is the same as the list of remainders $[x_i]$ so that `CRT_solve` can be applied.

5. Show the Pairwise Coprimality of Moduli

```coq
(* 5. Pairwise coprimality of moduli (from
    factor_moduli_coprime). *)
assert (HCoprime : forall i j, i <> j ->
  Nat.gcd (nth i mods 1) (nth j mods 1) = 1).
```

```
{
  intros i j Hij.
  exact (factor_moduli_coprime n facs mods eq_refl
      eq_refl i j Hij).
}
```

Here, we use FTOA oracle to ensure that any two distinct primepower factors are coprime. In particular, $\gcd(p_i^{c_i}, p_j^{c_j}) = 1$ for $i \neq j$.

Assemble CRT and Define x

```
(* Use CRT_solve to get x < n with x     x_i (mod p^k).
   *)
specialize (CRT_solve_correct mods rems Hlen HCoprime)
    as [Hx_bound Hcong].
set (x := CRT_solve mods rems).
```

This code invokes the CRT axiom to produce a single integer x with the property $x \bmod p_i^{c_i} = x_i$ for each i. We also record that $x < \prod_i p_i^{c_i} = n$.

6. Define $D\text{gen}^x \cdot h^{-1}$, then show that $D = e$.

```
(* 6. Let D := gen^x * (inv h). Show D = e. *)
set (D := op (pow gen x) (inv h)).

(* 6a. Prove for each (p,k) in facs: pow D (n/m) = e. *)
assert (HD_ord : forall p k, In (p, k) facs ->
  let m := p ^ k in
  pow D (n / m) = e).
{
  intros p k Hin.
  set (m := p ^ k).

  (* Find i so that nth_error facs i = Some (p,k). *)
  apply in_nth_error in Hin as [i Hnth_err].

  (* Define x_i     *)
  set (x_i := discrete_log_primepower (pow gen (n / m))
    (pow h (n / m))
    p k).

  subst rems.
  (* Show   nth   i rems 0 =  x _ i  . *)
  assert (Hrem_eq : nth i rems 0 = x_i).
  {
    revert i Hnth_err.
```

COQ

```
    induction facs as [| [p0 k0] rest IH]; intros i Hnth_err.
    - (* facs = []: impossible *)
      simpl in Hnth_err.
      rewrite nth_error_nil in Hnth_err.
      discriminate Hnth_err.
    - (* facs = (p0,k0)::rest *)
      destruct i as [| i'] eqn:Ei.
      + (* i = 0 *)
        simpl in Hnth_err. inversion Hnth_err; subst.
        simpl. unfold x_i, m. reflexivity.
      + (* i = S i' *)
        simpl in Hnth_err. apply IH. assumption.
  }

  (* From CRT_solve_correct: x mod m = x_i. *)
  assert (Hmod : x mod m = x_i).
  {
    rewrite Hrem_eq; clear Hrem_eq.
    rewrite <- (proj2 (proj2 (CRT_solve_correct mods rems Hlen
      HCoprime))) by auto.
    assert (Hmods_eq : nth i mods 1 = m).
    { subst mods. rewrite <- (nth_error_nth _ _ 1 Hnth_err).
      reflexivity. }
    rewrite Hmods_eq. reflexivity.
  }

  (* Now compute (gen^x)^(n/m) = (gen^(n/m))^{x mod m} = *)
  rewrite <- pow_mul. rewrite Hmod.
  rewrite Nat.mul_add_distr_l. rewrite <- pow_add.


  (* Since (m * (n/m)) = n by factor_divides: *)
  pose proof (factor_divides n p k Hin) as [Hmn_nz Hmn_div].
  rewrite Nat.mul_comm in Hmn_div. rewrite Hmn_div.

  (* Replace (gen^n = e) *)
  rewrite pow_mul. rewrite gen_order. rewrite id_r.

  (* (inv h)^(n/m) = inv (h^(n/m)) *)
  assert (Inv_pow : forall y0 k0, pow (inv y0) k0 = inv (pow y0 k0)).
  intros y0 k0. induction k0 as [| k0' IHk].
  - simpl. rewrite pow0, pow0, id_r. reflexivity.
```

```
67    - simpl. rewrite powS, IHk. rewrite <- inv_l, <- assoc
          .
68      reflexivity.
69    rewrite pow_add. rewrite Inv_pow.
70
71    (* Use Hpp: (gen^(n/m))^{x_i} = h^{(n/m)} *)
72    specialize (Hpp i p k Hnth_err) as [Hx_lt Hgi_eq].
73    rewrite <- pow_mul in Hgi_eq.
74    replace (pow gen (x_i * (n / m))) with (pow (pow gen (
          n / m)) x_i).
75    { rewrite Hgi_eq. rewrite id_l. reflexivity. }
76    apply pow_mul.
77  }
```

Here, we verify that for each prime power $p_i^{c_i}$, the element $D = \text{gen}^x \cdot h^{-1}$ satisfies $D^{n/m_i} = e$. After showing that the ith entry of `rems` is equal to x_i, we combine the results:

$$\text{gen}^{(n/m_i)x_i} = h^{(n/m_i)} \ \wedge \ \text{gen}^n = e \ \Rightarrow \ (\text{gen}^x h^{-1})^{n/m_i} = e.$$

7. Conclude that indeed, $D = e$.

```
1   (* 7. By cyclic_surjection, write D = gen^y for some y <
        n. *)
2   destruct (cyclic_surjection D) as [y [Hy_lt HD_y]].
3
4   (* From HD_ord: (gen^y)^(n/m) = e, so p^k | y. *)
5   assert (Hdiv_factor : forall p k, In (p, k) facs -> (p ^
        k) | y).
6   {
7     intros p k Hin.
8     specialize (HD_ord p k Hin) as Hpow_e. simpl in Hpow_e
          .
9     pose proof
10      (primepower_divides_exponent
11        y p k
12        (proj2 (factor_divides n p k Hin))
13        Hpow_e) as Hm_div.
14    exact Hm_div.
15  }
16
17  (* Since Product p^k = n and each p^k | y, we get n | y.
        *)
18  assert (Hn_div_y : n | y).
19  apply (product_of_factors_divides facs y); [ exact Hprod
        | exact Hdiv_factor ].
20
```

```
(* But y < n, so y = 0 (by small_multiple_zero), hence D
    = e. *)
pose proof (small_multiple_zero y Hy_lt Hn_div_y) as Hy0
    . subst y.
rewrite pow0 in HD_y. rewrite id_l in HD_y. clear HD_y.
```

Here, we utilize the cyclicity to express $D = \text{gen}^y$. In addition, since $D^{n/p_i^{c_i}} = e$ for any i, then we deduce that y is divisible by each prime power, so $n \mid y$. And now, since $y < n$, we are forced that $y = 0$, concluding that $D = \text{gen}^0 = e$.

8. Conclude that $\text{gen}^x = h$.

```
1 (* 8. Finally, D = (gen^x)*(inv h) = e implies gen^x =
    h. *)
2 unfold D. rewrite <- id_l.
3 apply (f_equal (fun z => op z h)). rewrite assoc,
    inv_l, id_l.
4 reflexivity.
5 Qed.
```

Since $D \cdot h = e \cdot h$ and we know that h^{-1} exists, we finally conclude that $\text{gen}^x = h$. This completes the proof of correctness: applying `discrete_log` to any $h \in G$ yields an x such that $\text{gen}^x = h$ via the Pohlig–Hellman Algorithm.

4 Conclusions and Future Works

This paper presented the formal verification of the Pohlig–Hellman algorithm using the Coq proof assistant. Through this formalization, the algorithm's correctness in solving the Discrete Logarithm Problem (DLP) within finite cyclic groups of known order was rigorously established. By introducing an abstract finite cyclic group structure, this development allows the separation of the algorithmic logic from the algebraic assumptions, allowing the proof to generalize beyond modular arithmetic to any cyclic group satisfying the given axioms. The proof leveraged a number of foundational theorems such as the Fundamental Theorem of Arithmetic, Fermat's Little Theorem, and the Chinese Remainder Theorem, all trusted axioms, enabling the reconstruction of the algorithm's decomposition and recomposition process.

The formal verification confirmed that the Pohlig–Hellman method indeed computes the correct discrete logarithm for all valid inputs using its mathematical assumptions. More broadly, this work demonstrates how Coq can effectively capture nontrivial cryptographic reasoning, ensuring correctness beyond empirical testing. This machine checked approach enhances confidence in both the theoretical soundness and practical implementation of cryptographic algorithms.

Additionally, the verification process revealed the algorithm's sensitivity to group orders with small prime factors. This finding emphasizes the importance of

using subgroups of large prime order in cryptographic applications, since small factors can make the discrete logarithm easier to compute and thus weaken overall security.

Future research may also extend the verification to include the axioms presented in the proof of the Pohlig–Hellman method. This involves formalizing the Fundamental Theorem of Arithmetic, Fermat's Little Theorem, and the Chinese Remainder Theorem within Coq rather than assuming them as axioms. Doing so would yield a self-contained and more complete formal proof of the algorithm. Another possible direction is to analyze the computational complexity of the Pohlig–Hellman encryption algorithm to evaluate its efficiency, speed, and computational cost in practice.

References

1. Teske, E.: The Pohlig–Hellman method generalized for group structure computation. J. Symbolic Comput. 27(6), 521–534 (1999). Elsevier
2. Fontein, F.: Groups from cyclic infrastructures and Pohlig–Hellman in certain infrastructures. arXiv preprint arXiv:0803.2123 (2008)
3. Sommerseth, M.L., Høiland, H.: Pohlig–Hellman applied in elliptic curve cryptography. Technical report, University of California Santa Barbara (2015)
4. Rahim, R.: Applied Pohlig–Hellman algorithm in three-pass protocol communication. J. Appl. Eng. Sci. 16(3) (2018)
5. Pohlig, S.C., Hellman, M.E.: An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. In: *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*, pp. 415–430 (2022)
6. Granger, R., Kleinjung, T., Zumbrägel, J.: On the discrete logarithm problem in finite fields of fixed characteristic. Trans. Amer. Math. Soc. 370(5), 3129–3145 (2018)
7. Math Center, Oxford/Emory: The Fundamental Theorem of Arithmetic. Available online at: https://mathcenter.oxford.emory.edu/site/math125/fundThmArithmetic/ (Accessed 2025)
8. Min-Ru, L.: The Fundamental Theorem of Arithmetic. Department of Mathematics, University of Houston. Available online at: https://www.math.uh.edu/ minru/spring11/fundamental-theorem.pdf (Accessed 2025)
9. Bishop, R.E.: On Fermat's Little Theorem. Department of Mathematics, University of Chicago (VIGRE REU). Available online at: https://www.math.uchicago.edu/ may/VIGRE/VIGRE2008/REUPapers/Bishop.pdf (Date: July 15, 2008; Accessed 2025)
10. Conrad, K.: Fermat's Little Theorem. Department of Mathematics, University of Connecticut. Available online at: https://kconrad.math.uconn.edu/blurbs/ugradnumthy/fermatlittletheorem.pdf (Accessed 2025)
11. Lynn, B.: The Chinese Remainder Theorem. Stanford University (PBC Number Theory Notes). Available online at: https://crypto.stanford.edu/pbc/notes/numbertheory/crt.html (Accessed 2025)
12. Conrad, K.: The Chinese Remainder Theorem. Department of Mathematics, University of Connecticut. Available online at: https://kconrad.math.uconn.edu/blurbs/ugradnumthy/crt.pdf (Accessed 2025)
13. S. C. Pohlig and M. E. Hellman, "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance," *IEEE Transactions on Information Theory*, vol. 24, no. 1, pp. 106–110, Jan. 1978.
14. The Coq Development Team, "The Coq Proof Assistant — Reference Manual, version 8.20.0," Sep. 2024. https://zenodo.org/records/14542673/files/coq-8.20.0-reference-manual.pdf
15. B. C. Pierce et al., *Software Foundations*, Vol. 1: Logical Foundations, Electronic textbook; University of Pennsylvania, ongoing. Available at: https://softwarefoundations.cis.upenn.edu/
16. A. Mahboubi and E. Tassi (with contributions by Y. Bertot, G. Gonthier, ...), *Mathematical Components*, MathComp project (book and libraries), 2016–. Available at: https://math-comp.github.io/mcb/

17. G. Gonthier, "A computer-checked proof of the four-colour theorem," Microsoft Research / INRIA report, 2005. Available (pdf) at: https://www.cse.chalmers.se/~abela/lehre/WS05-06/CAFR/4colproof.pdf
18. G. Barthe, B. Grégoire, S. Hilaire, J. Kelsey, B. Köpf, and B. M. yk, "Formal certification of code-based cryptographic proofs," in *Proceedings of the 2009 ACM SIGPLAN International Conference on Functional Programming (POPL / related venues)*, 2009. Available: https://www-sop.inria.fr/members/Benjamin.Gregoire/Publi/popl09.pdf.
19. A. Petcher and G. Morrisett, "The Foundational Cryptography Framework," arXiv preprint, 2014 (conference version 2015). Available: https://arxiv.org/abs/1410.3735.
20. A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple high-level code for cryptographic arithmetic — with proofs, without compromises," in *IEEE Symposium on Security and Privacy*, 2019. Available: https://jasongross.github.io/papers/2019-fiat-crypto-ieee-sp.pdf.
21. EasyCrypt Development Team, "EasyCrypt: Computer-Aided Cryptographic Proofs," project web page / repository. Available: https://github.com/EasyCrypt/easycrypt.
22. J. Protzenko, A. M. Kovich, B. Parno, et al., "EverCrypt: A fast, verified, cross-platform cryptographic provider," (Everest / EverCrypt project), 2019. Available: https://www.andrew.cmu.edu/user/bparno/papers/evercrypt.pdf.
23. HACL* / Project Everest, "HACL*: a formally verified cryptographic library (F* / Low*)," project web page. Available: https://hacl-star.github.io/.
24. "Verification of a Cryptographic Primitive: SHA-256," verifiable-C / separation-logic style development (representative paper demonstrating verified primitive implementations in Coq/VST). Available: https://dl.acm.org/doi/10.1145/2701415.