Cascader: A Recurrence-Based Key Exchange Protocol

Anders Lindman

Independent Researcher anders_lindman@yahoo.com

Abstract. Cascader, a novel key-exchange protocol based on an iterative multiplicative recurrence over a finite field, is introduced. In contrast to standard methods, e.g., traditional Diffie–Hellman and ECC, it replaces exponentiation and scalar multiplication with layered products, achieving commutativity and deterministic pseudorandom behavior.

1 Introduction

Cryptography has long relied on number-theoretic hardness assumptions like the discrete logarithm problem. In contrast, this paper explores a new approach: a key exchange built from a structured recurrence. The core idea is inspired by iterated multiplicative transformations that resemble PRNG-like behavior. The result is a system where two parties can arrive at the same shared secret without direct key exchange — but using entirely different mathematics. We call this method `Cascader` due to its cascading product structure and layered computation.

2 Definition of the Recurrence

Let p be a large prime modulus (e.g. $p = 2^{256} - 189$). For any non-negative integer e, let

$$e = \sum_{k=0}^{m-1} b_k 2^k, \qquad b_k \in \{0, 1\}$$

denote its binary expansion, and let

$$\mathcal{I}(e) = \{k + 1 \mid b_k = 1\}$$

be the set of *1-based* indices of the set bits of e. Define

$$F(x, e) = x \cdot \prod_{j \in \mathcal{I}(e)} \left(3^j \cdot 2^{\frac{j(j-1)}{2}}\right) \mod p.$$

Equivalently, using the Kronecker delta,

$$F(x, e) = x \cdot \prod_{k=0}^{m-1} \left(3^{k+1} \cdot 2^{\frac{k(k+1)}{2}}\right)^{b_k} \mod p \tag{1}$$

This function is deterministic, modular, and composable, allowing two parties to compute the same shared secret when used as described in the protocol.

3 Choice of Constant and Modulus

This section explains the pragmatic reasons for selecting the multiplier 3 and the prime modulus:

$$MOD = 2^{256} - 189.$$

3.1 Multiplier 3

A small odd integer c is needed that is easy to multiply and at the same time avoids trivial short cycles in the recurrence:

$$x_{n+1} = (c \cdot x_n) \bmod MOD.$$

- 3 is the smallest odd prime larger than 1, giving a low Hamming weight and an efficient implementation.
- Because $\gcd(3, MOD) = 1$, the multiplier is invertible, so the map is a permutation of $\mathbb{Z}^*_{MOD}$.
- Wolfram Alpha[1] confirms that 3 is a primitive root modulo MOD.

3.2 Modulus $2^{256} - 189$

The prime:

$$p = 2^{256} - 189$$

was selected as p is very close to 2^{256}, making modular reductions exploit the size of the machine word, and because the multiplier 3 is a primitive root modulo p.

3.3 Cardinality of the Output Space

Let the private exponent e be a 256-bit integer. The size of the input space (the number of possible private keys) is therefore 2^{256}. The outputs of the function $F(\text{SEED}, e)$ are elements of the multiplicative group $\mathbb{Z}^*_p$, which has a size of $p - 1 = 2^{256} - 190$.

Pigeonhole Principle and Inevitable Collisions. Since the size of the input space (2^{256}) is larger than the size of the output space ($2^{256} - 190$), the pigeonhole principle dictates that the function $e \mapsto F(\text{SEED}, e)$ cannot be injective. There must be collisions; that is, there must exist distinct private keys $e_1 \neq e_2$ such that $F(\text{SEED}, e_1) = F(\text{SEED}, e_2)$. At least 190 such collisions are guaranteed to exist across the entire key space.

Expected Coverage. While perfect injectivity is impossible, a desirable cryptographic property is that the function's output 'image' covers a large fraction of the codomain. If we heuristically model F as a random function mapping inputs to outputs, the number of distinct output values is expected to be very close to the size of the output space, $p-1$. The key security assumption is that finding such collisions is computationally infeasible. The existence of a small number of guaranteed collisions over a vast key space does not, in itself, constitute a practical attack, though it underscores the importance of the collision resistance property discussed in Section 8.

3.4 The primitive-root property of 3

Having 3 being a primitive root modulo p does not force the full Cascader recurrence to cycle through all $p-1$ residues. Nevertheless, the property still yields useful guarantees.

- **Invertibility.** The single-step map $x \mapsto 3x \bmod p$ is a permutation of $\mathbb{F}_p^*$; hence every partial product inside the recurrence is non-zero and invertible.
- **Large order of factors.** Each factor $3^j \cdot 2^{j(j-1)/2}$ has order at least that of 3, ensuring that the contribution of every bit position is highly non-trivial.
- **Empirical uniformity.** Preliminary statistical tests show that the final outputs are uniform, although the exact period remains unknown.

In short, the primitive-root status of 3 guarantees strong diffusion without promising a full period, and empirical evidence supports the practical adequacy of the choice.

3.5 Summary

The pair $(c, MOD) = (3, 2^{256} - 189)$ was chosen as adequate for a proof-of-concept. Future work will investigate alternative small multipliers and formally bound the statistical quality of the resulting sequences.

4 Pseudocode Description

Below is the algorithmic description of the key exchange function:

5 Key Exchange Protocol

`Cascader` works as follows:

- Alice computes her public key: $A = F(SEED, a)$
- Bob computes his public key: $B = F(SEED, b)$
- Alice computes shared secret: $S_A = F(B, a)$
- Bob computes shared secret: $S_B = F(A, b)$

From the structure of $F(x, e)$, we observe empirically:

Algorithm 1 `linearRecurrence`(base, exponents)

```
Require: MOD is a global constant modulus
 1: $result \leftarrow base$
 2: $exp \leftarrow 1$
 3: while $exponents > 0$ do
 4:     if $exponents \mod 2 = 1$ then
 5:         $mult \leftarrow 1$
 6:         for $i \leftarrow 0$ to $exp - 1$ do
 7:             $result \leftarrow (3 \cdot result \cdot mult) \mod MOD$
 8:             $mult \leftarrow mult \ll 1$                ▷ Multiply by 2 via bit shift
 9:     $exponents \leftarrow exponents \gg 1$
10:     $exp \leftarrow exp + 1$
11: return $result$
```

$$F(F(SEED, a), b) = F(F(SEED, b), a)$$

This allows both parties to compute the same shared secret, enabling secure communication.

6 Time Complexity Analysis

Let n denote the bit length of the private exponent (here $n = 256$) and let b be its Hamming weight.

Loop structure. The outer `while` loop iterates once for every bit of the exponent, yielding $\Theta(n)$ iterations. Inside this loop, a set bit at 1-based position j triggers an inner loop that runs j times. The total number of inner-loop iterations is therefore the sum of the positions of the set bits in the exponent, $\sum_{j \in \mathcal{I}(e)} j$.

Expected cost. For a uniformly random n-bit key, the expected Hamming weight is $\mathbb{E}[b] = \frac{n}{2}$, and the set bits are distributed across the range $[1, n]$. The expected sum of bit positions is $\sum_{j=1}^{n} j \cdot \mathbb{P}(\text{bit } j - 1 \text{ is set}) = \sum_{j=1}^{n} \frac{j}{2} = \frac{n(n+1)}{4}$. The complete running time is therefore dominated by the inner-loop multiplications, leading to an expected-time complexity of:

$$\mathbb{E}[T(n)] = \Theta(n^2).$$

For $n = 256$, this results in a notable number of modular multiplications, confirming that the protocol is computationally intensive.

7 Commutativity Proof

We now prove that the protocol correctly establishes a shared secret, i.e., that $S_A = S_B$. The property relies on the commutativity of modular multiplication.

Let the shared base be $x = \text{SEED}$, and let Alice's and Bob's private keys be a and b respectively. Let their corresponding 1-based index sets be $\mathcal{I}(a)$ and $\mathcal{I}(b)$.

Define the product factor for an exponent e as:

$$P(e) = \prod_{j \in \mathcal{I}(e)} C_j \pmod{p}, \quad \text{where} \quad C_j = 3^j \cdot 2^{\frac{j(j-1)}{2}}$$

From the definition in Section 2, the function F can be written as $F(x, e) = x \cdot P(e) \pmod{p}$.

Alice computes her public key $A = F(x, a) = x \cdot P(a) \pmod{p}$. Bob computes his public key $B = F(x, b) = x \cdot P(b) \pmod{p}$.

Next, they compute the shared secret:

- Alice computes $S_A = F(B, a) = F(x \cdot P(b), a) = \big(x \cdot P(b)\big) \cdot P(a) \pmod{p}$.
- Bob computes $S_B = F(A, b) = F(x \cdot P(a), b) = \big(x \cdot P(a)\big) \cdot P(b) \pmod{p}$.

Since multiplication in the finite field $\mathbb{Z}_p$ is associative and commutative, we have:

$$S_A = x \cdot P(b) \cdot P(a) = x \cdot P(a) \cdot P(b) = S_B$$

This proves that both parties arrive at the identical shared secret, so the protocol is correct.

8 Security Intuition

The security of the proposed key exchange algorithm, similarly to established public-key cryptosystems like Diffie-Hellman, fundamentally relies on the computational difficulty of certain number-theoretic problems. Our design aims to construct a function that behaves as a *one-way function*—easy to compute in one direction but computationally infeasible to reverse—while also ensuring strong *collision resistance*.

8.1 Commutativity for Shared Secret Derivation

The functional correctness of the key exchange, ensuring that Alice and Bob derive the same shared secret, stems from the commutative property of the underlying modular multiplication. Let $F(\text{base}, e)$ be the function that generates a public key from a given base and a private exponent e. Our construction yields a property analogous to $F(X, Y) = X \cdot P(Y) \pmod{p}$, where $P(Y)$ is a factor derived from Y. Given this, the shared secret derivation is inherently commutative:

$$\begin{aligned} \text{SharedSecret} &= F(F(\text{SEED}, \text{alicePrivate}), \text{bobPrivate}) \\ &= F(F(\text{SEED}, \text{bobPrivate}), \text{alicePrivate}) \end{aligned}$$

This ensures that the protocol achieves its functional goal regardless of other security properties.

8.2 One-Wayness and Injectivity

For the algorithm to be cryptographically secure, merely achieving functional correctness is insufficient. The function $F(\text{SEED}, e)$ must also exhibit properties that prevent an adversary (Eve) from deriving secret information from publicly available data.

One-Way Function Property The primary security pillar is that $F(\text{SEED}, e)$ must act as a *one-way function*. That is, given the public base SEED and the public key $F(\text{SEED}, e)$, it should be computationally infeasible for Eve to recover the private exponent e. This difficulty is analogous to the Discrete Logarithm Problem (DLP) which underpins the security of classic Diffie-Hellman, where it is hard to find x given $g^x \pmod p$. If F were easily invertible, Eve could determine Alice's private key from her public key, thus immediately computing the shared secret.

Injectivity and Collision Resistance Beyond one-wayness, the design strives for strong *injectivity*. This means that each distinct private exponent e should ideally map to a unique public key $F(\text{SEED}, e)$. In other words, if $e_1 \neq e_2$, then $F(\text{SEED}, e_1) \neq F(\text{SEED}, e_2)$. The failure to achieve injectivity implies the existence of *collisions*, where $F(\text{SEED}, e_1) = F(\text{SEED}, e_2)$ for $e_1 \neq e_2$. Our design attempts to establish this injectivity and resistance to collisions through the carefully constructed multiplicative factors C_j:

$$C_j = 3^j \cdot 2^{j(j-1)/2} \pmod p$$

The use of polynomially varying exponents (j and $j(j-1)/2$) for each bit position j was intended to create a unique "signature" or "contribution" for each bit. This aims to enforce a form of *multiplicative independence* among the C_j values, such that no non-trivial subset of C_j factors would multiply to $1 \pmod p$. Conceptually, this is akin to ensuring that the contributions of each bit are "orthogonal" in the multiplicative group, making it exceptionally difficult for different combinations of bits (private keys) to yield the same public key.

8.3 The Challenge of Structured Hardness

While the intention behind the structured exponents is to guarantee distinctness and prevent trivial collisions, the very presence of this mathematical structure presents a significant cryptographic challenge. Unlike problems based on purely random elements, the predictable relationship within C_j may be exploitable by advanced cryptanalytic techniques, such as those involving lattice reduction algorithms. These attacks aim to find non-trivial linear (or, in this case, exponent-based) relations among the structured elements that sum to zero, which could correspond to finding collisions or inverting the function.

8.4 Empirical Observations and Formal Proofs

Initial empirical tests, such as plotting the output distributions and standard statistical randomness tests (e.g., Monobit and Runs tests), show that the function produces outputs that are statistically indistinguishable from random noise. For instance, in a sample of 1,000,000 bits:

- **Monobit Test:** ones: 499,950; zeros: 500,050; χ^2 p-value: 0.9950
- **Runs Test:** observed runs: 499,672; expected runs: 500,001.0; z-score: -0.658
- **Serial Correlation:** flip rate: 0.4997 (ideal 0.500)

These results confirm the excellent statistical properties and uniformity of the generated sequence. However, it is crucial to emphasize that passing statistical randomness tests, while necessary, does not directly imply cryptographic unpredictability or resistance to specific structural attacks. The true security relies on the unproven computational hardness of the underlying mathematical problem, specifically the hardness of inverting $F(\text{SEED}, e)$ or finding collisions, given the structured nature of the C_j terms. A rigorous cryptographic proof of this hardness against all known attacks remains an essential area for future work.

9 Security Analysis and Proof Status

9.1 Proof of Correct Functionality

Definition 1 (Correctness). *The* Cascader key-exchange protocol *is* correct *if for all private exponents $a, b \in \{0,1\}^{256}$ the two parties compute the same shared secret:*

$$F\big(F(SEED, a),\, b\big) \;=\; F\big(F(SEED, b),\, a\big).$$

Theorem 1 (Commutativity). *Cascader is correct in the sense of Definition 1.*

Proof. Immediate from the commutativity of modular multiplication in $\mathbb{Z}_p$ and the structure of the product in (1), as formally shown in Section 7.

9.2 Security Reduction to Discrete Logarithm

We now present a security argument suggesting that the function $F(x, e)$ used in the key exchange protocol is at least as hard to invert as the standard modular exponentiation function, whose security rests on the well-known discrete logarithm problem (DLP).

Function Definition Let $p = 2^{256} - 189$ be a prime modulus. Define the function:

$$F(x, e) = x \cdot \prod_{k=0}^{n} \alpha_k^{b_k} \mod p,$$

where $e = \sum_{k=0}^{n} b_k 2^k$ is the binary representation of e, and:

$$\alpha_k = 3^{k+1} \cdot 2^{\frac{k(k+1)}{2}} \mod p.$$

Each bit $b_k \in \{0, 1\}$ determines whether the term α_k is included in the product. The base value x (e.g., the shared SEED) is multiplied into the final result.

Modular Exponentiation as a Special Case Consider the standard modular exponentiation function:

$$\text{ModExp}(g, e) = g^e \mod p.$$

We now show that modular exponentiation is a special case of $F(x, e)$.

Let $x = 1$ and choose $\alpha_k = g^{2^k} \mod p$ for some generator $g \in \mathbb{Z}_p^*$. Then:

$$F(1, e) = \prod_{k=0}^{n} \alpha_k^{b_k} = \prod_{k=0}^{n} (g^{2^k})^{b_k} = g^{\sum_{k=0}^{n} b_k 2^k} = g^e \mod p.$$

Thus, modular exponentiation is a special instance of F under specific α_k values.

Reduction Argument Suppose there exists an efficient algorithm $\mathcal{A}$ that, given x and $F(x, e)$, can recover e.

Then, for any modular exponentiation instance $g^e \mod p$, we can define:

$$\alpha_k = g^{2^k} \mod p, \quad x = 1.$$

So $F(1, e) = g^e \mod p$, and we can recover e using $\mathcal{A}$. Hence, $\mathcal{A}$ would solve the discrete logarithm problem.

This contradicts the assumed hardness of DLP, which underlies the security of the Diffie–Hellman key exchange and other cryptographic systems.

Strength of Actual $\boldsymbol{\alpha_k}$ In our construction, each α_k is defined as:

$$\alpha_k = 3^{k+1} \cdot 2^{\frac{k(k+1)}{2}} \mod p,$$

where 3 is a primitive root modulo p, and the exponential growth of both components increases entropy and resists optimizations that exploit structure (e.g., lattice attacks, meet-in-the-middle).

This additional complexity makes the inversion problem potentially harder than standard DLP.

Conclusion Since modular exponentiation is a special case of $F(x, e)$, and the actual structure of F introduces additional multiplicative complexity and obfuscation via α_k, we conclude:

> Inverting $F(x, e)$ is at least as hard as solving the discrete logarithm problem in $\mathbb{Z}_p^*$. Therefore, F inherits the one-wayness and security properties of modular exponentiation.

9.3 Heuristic Arguments in the Random-Oracle Model

Assumption 1 (Cascader). *The function SEED $\mapsto F(SEED, e)$ behaves like a random oracle: for any $e \neq 0$ the value $F(SEED, e)$ is uniformly distributed in $\mathbb{Z}_p$ and independent of all other queries.*

Theorem 2 (ROM security). *Under Assumption 1, the Cascader key exchange is indistinguishable from an ideal key-exchange protocol in the random-oracle model.*

Proof (Proof sketch). The shared key $S = F\big(F(\mathrm{SEED}, a), b\big) = F\big(F(\mathrm{SEED}, b), a\big)$ is the output of the random oracle on the unique input (SEED, a, b); hence it is uniform and independent of the public keys A and B.

Remark 1. Theorem 2 is *not* a reduction, because Assumption 1 itself lacks justification from a standard hard problem.

Feature	**Cascader**	**Modular Exponentiation (DH)**
Core Operation	Layered product recurrence	$g^a \bmod p$
Complexity (n=bit length)	$O(n^2)$	$O(n)$
Known Hardness Assumption	None (Novel Problem)	Discrete Log Problem
Inversion Method	No known shortcut	Well-known algorithms
Shared Secret Agreement	Yes	Yes
Interoperability	No	No
Novelty	High	Standard

Table 1. Comparison of Cascader with traditional modular exponentiation.

Aspect	**Cascader**	**ECC (X25519)**
Primitive	Iterative product	Scalar mult. on curve
Hard problem	None (Novel Problem)	ECDLP / ECDH
Key size	256-bit priv. / 256-bit pub.	256-bit priv. / 256-bit pub.
Compute cost (n=bit length)	$\Theta(n^2)$	$\Theta(n)$
Side-channel	High (bit-dependent loop)	Constant-time (Montgomery ladder)
Hardware accel.	Future applicability	Widely available
Standardisation	None	RFC 7748, TLS 1.3, WireGuard

Table 2. Cascader versus X25519 for 128-bit classical security.

Summary Cascader's merit is the novelty of the construction and the potential for different security aspects compared to know key exchange protocols. Potential downsides include: slower than standard key exchange protocols, lack of a recognized hardness assumption, and no constant-time guarantees.

10 Open Problems and Future Work

Here are several directions for future exploration:

1. Can this recurrence be inverted efficiently?
2. What is the entropy and coverage of the output space?
3. Is it resistant to cryptographic attacks or side-channel analysis?
4. Can it be used as a slow KDF or commitment scheme?
5. Can it be optimized using precomputed layers or parallelization?
6. Could it form the basis of a hash chain or proof-of-work system?

11 Conclusion

We introduced Cascader, a recurrence-based key exchange protocol that constitutes a novel approach compared to standard cryptographic primitives. While not aimed at performance-critical environments, it opens up new directions in public-key protocols.

References

1. Wan Mohd Rosly, Wan Nur Shaziayani, Sharifah Sarimah Syed Abdullah, and Fuziatul Norsyiha Ahmad Shukri. *The uses of Wolfram Alpha in mathematics. Teaching and Learning in Higher Education (TLHE)*, vol. 1, 2020, pp. 96–103.

A Reference Implementation in JavaScript

Below is a JavaScript implementation of the recurrence-based key exchange algorithm described in this paper. This reference code demonstrates the core functions for the protocol.

```
const KEY_SIZE_BITS = 256n;
const MAX_INT = 1n << KEY_SIZE_BITS;
const MOD = MAX_INT - 189n; // Prime number
const SEED = MAX_INT / 5n;

function linearRecurrence(seed, exponents) {

  let result = seed;
  let exp = 1n;

  while (exponents > 0n) {
    if (exponents % 2n === 1n) {
      let mult = 1n;
      for (let i = 0; i < exp; i++) {
        result = 3n * result * mult % MOD;
        mult <<= 1n;
      }
    }
    exponents >>=  1n;
    exp++;
  }
  return result;
}

// Generate a random 256-bit BigInt
function random256BitBigInt() {
  const array = new Uint8Array(32);
  crypto.getRandomValues(array);
  let hex = '0x';
  for (const byte of array) {
    hex += byte.toString(16).padStart(2, '0');
  }
  return BigInt(hex);
}

const alicePrivate = random256BitBigInt();
const bobPrivate = random256BitBigInt();
const alicePublic = linearRecurrence(SEED, alicePrivate);
const bobPublic = linearRecurrence(SEED, bobPrivate);
const aliceShared = linearRecurrence(bobPublic,
    alicePrivate);
const bobShared = linearRecurrence(alicePublic, bobPrivate);
```

```
console.log("Alice private", alicePrivate.toString());
console.log("Bob private", bobPrivate.toString());
console.log("Alice public", alicePublic.toString());
console.log("Bob public", bobPublic.toString());
console.log("Alice Shared", aliceShared.toString());
console.log("Bob Shared", bobShared.toString());
console.log("Alice's and Bob's shared secrets equal?",
    aliceShared === bobShared);
```

Listing 1.1. Recurrence-Based Key Exchange in JavaScript