Solving the Elliptic Curve Discrete Logarithm Problem (ECDLP)

Analysis, Implementation, and Security Quantification

Team : *StalinSort in O(5)*

Members :

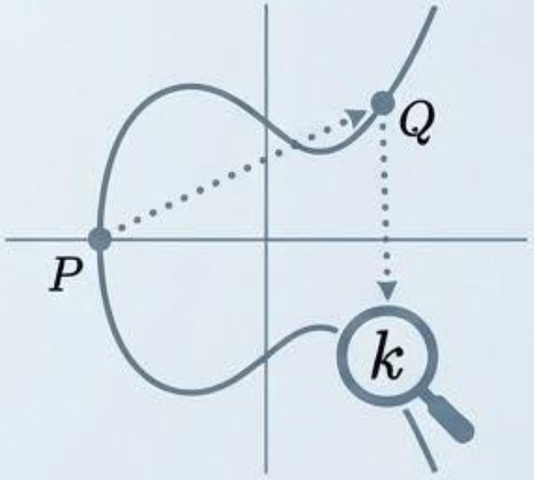
1. Hemang Joshi
2. Amish Goyal
3. Shardul Joshi
4. Kausheya Roy
5. Harith Yerragolam

Formal Problem Definition & Motivation

Formal Definition of ECDLP

Given an elliptic curve **E(Fp)** defined over a finite field Fp, a generator point **P** on the curve, and a target point **Q** on the curve such that **Q = kP**, the Elliptic Curve Discrete Logarithm Problem (ECDLP) is to find the integer **k**, where **k** is the discrete logarithm of **Q** with respect to **P**. This **k** is typically a secret integer within the range [1, n-1], where n is the order of the point P.

$$Q = kP$$



Real-World Relevance: Why ECDLP?

ECDLP forms the cryptographic hardness assumption underpinning Elliptic Curve Cryptography (ECC). Its computational intractability is crucial for securing a vast array of digital communications and transactions. ECC is widely adopted due to its superior security per bit compared to traditional cryptosystems like RSA.

- **TLS/HTTPS:** Securing web traffic for millions of websites.
- **ECDSA:** Digital signatures used in cryptocurrencies like Bitcoin and Ethereum.
- **Secure Messaging:** End-to-end encryption in popular messaging applications.
- **Key Exchange:** Establishing shared secrets over insecure channels (e.g., ECDH).

A 256-bit ECC key offers security roughly equivalent to a 3072-bit RSA key, leading to smaller key sizes, faster computations, and reduced bandwidth requirements.

Project Objectives

Implement and test multiple DLP-solving algorithms from scratch and measure run-time.

Analyze the impact of partial information leakage (LSB, interval, residues).

Validate theoretical complexity through experimental graphs.

Understand the specific scenarios where each algorithm becomes dominant.

High-Level Overview of ECDLP Algorithms

Brute Force

A straightforward linear search, iterating k from 1 until $kP = Q$. It serves as the baseline for performance comparison, with a time complexity of $O(n)$, where n is the order of the point P.

Baby-Step Giant-Step (BSGS)

A meet-in-the-middle algorithm that partitions the search space. It achieves $O(\sqrt{n})$ time complexity but requires $O(\sqrt{n})$ memory to store precomputed "baby steps." Important for illustrating the square-root barrier for generic DLP algorithms.

Pollard's Rho

A probabilistic algorithm that uses a pseudo-random walk to find collisions, leveraging Floyd's cycle-finding algorithm. It offers $O(\sqrt{n})$ time complexity with significantly improved $O(1)$ memory usage, making it practical for larger group orders than BSGS.

Pohlig–Hellman

This algorithm efficiently solves ECDLP when the order of the generator point n is a "smooth" number (i.e., it has only small prime factors). It breaks the original problem into smaller subproblems modulo each prime factor and combines the results using the Chinese Remainder Theorem. Crucial for understanding the impact of group order factorization on security.

Las Vegas Algorithms

These are randomized variants, primarily derived from Pollard's Rho or Kangaroo algorithms. While they always provide a correct answer, their runtime is probabilistic and varies. They often demonstrate faster average-case performance compared to their deterministic counterparts, especially in specific leakage scenarios.

Algorithmic Efficiency: Time and Space Complexity

A comparative overview of the theoretical performance characteristics for each implemented ECDLP-solving algorithm.

Brute Force	$O(n)$	$O(1)$	Direct, linear search, impractical for large n
BSGS	$O(\sqrt{n})$	$O(\sqrt{n})$	Meet-in-the-middle approach, memory-intensive
Pollard's Rho	$O(\sqrt{n})$ expected	$O(1)$	Randomized, highly memory-efficient
Pohlig–Hellman	$O(\Sigma \sqrt{p_i})$	$O(1)$	Optimal for smooth-order curves
Kangaroo	$O(\sqrt{(b-a)})$	$O(1)$	Efficient for bounded search intervals
Las Vegas	Expected $O(\sqrt{n})$	$O(1)$	Probabilistic, with runtime variance

This table highlights the fundamental trade-offs between computational time and memory requirements for each algorithm, guiding our empirical analysis.

Elliptic Curve Discrete Logarithm Problem (ECDLP): Algorithmic Space Complexity

The Elliptic Curve Discrete Logarithm Problem (ECDLP) is a cornerstone of modern cryptography. Efficiently solving for `k` in `Q = k·G` (where `G` is a base point and `Q` is another point on the elliptic curve) is computationally hard, forming the basis for secure communications. This presentation delves into the space complexity and data structures of various algorithms designed to tackle ECDLP, from the most naive brute force to more sophisticated approaches.

Shared Foundations: Core Data Structures Across All Algorithms

1

Point Representation

Elliptic curve points are fundamentally represented as an `Optional[Tuple[int, int]]`, where `None` denotes the point at infinity (the identity element). Affine coordinates `(x, y)` represent finite points on the curve. This consistent representation ensures interoperability across different algorithmic implementations.

2

EllipticCurve Class

A dedicated `EllipticCurve` class encapsulates the curve's defining parameters `(a, b, p)` (for a curve `y² = x³ + ax + b (mod p)`). It provides essential methods such as `add()` for point addition, `scalar_multiply()` for scalar multiplication, `negate()` for point negation, and `is_on_curve()` for validation. This object-oriented approach promotes code modularity and reusability.

3

Integer Arithmetic

All algorithms rely heavily on robust integer arithmetic within a finite field. Python's native integers support arbitrary precision, crucial for large cryptographic numbers. Modular inverse operations are typically performed using the Extended Euclidean Algorithm, while modular exponentiation leverages built-in functions like `pow()`. These low-level operations are optimized for performance, forming the bedrock of elliptic curve computations.

Brute Force: The Exhaustive Approach

Problem Solved: Finding the discrete logarithm 'k' where Q = kP

Working Procedure

Start with `k = 1`.

Increment `k` by `1` in each step.

Compute `R = kP`.

Compare `R` with `Q`.

Repeat until `kP = Q`.

Mathematical Foundation

The calculation `kP` is done via incremental addition:

$R_0 = P$

$R_i = R_{i-1} + P$

This process is deterministic and is guaranteed to find a solution because the ECDLP has a unique `k`.

Complexity & Properties

Time: *O(n)*

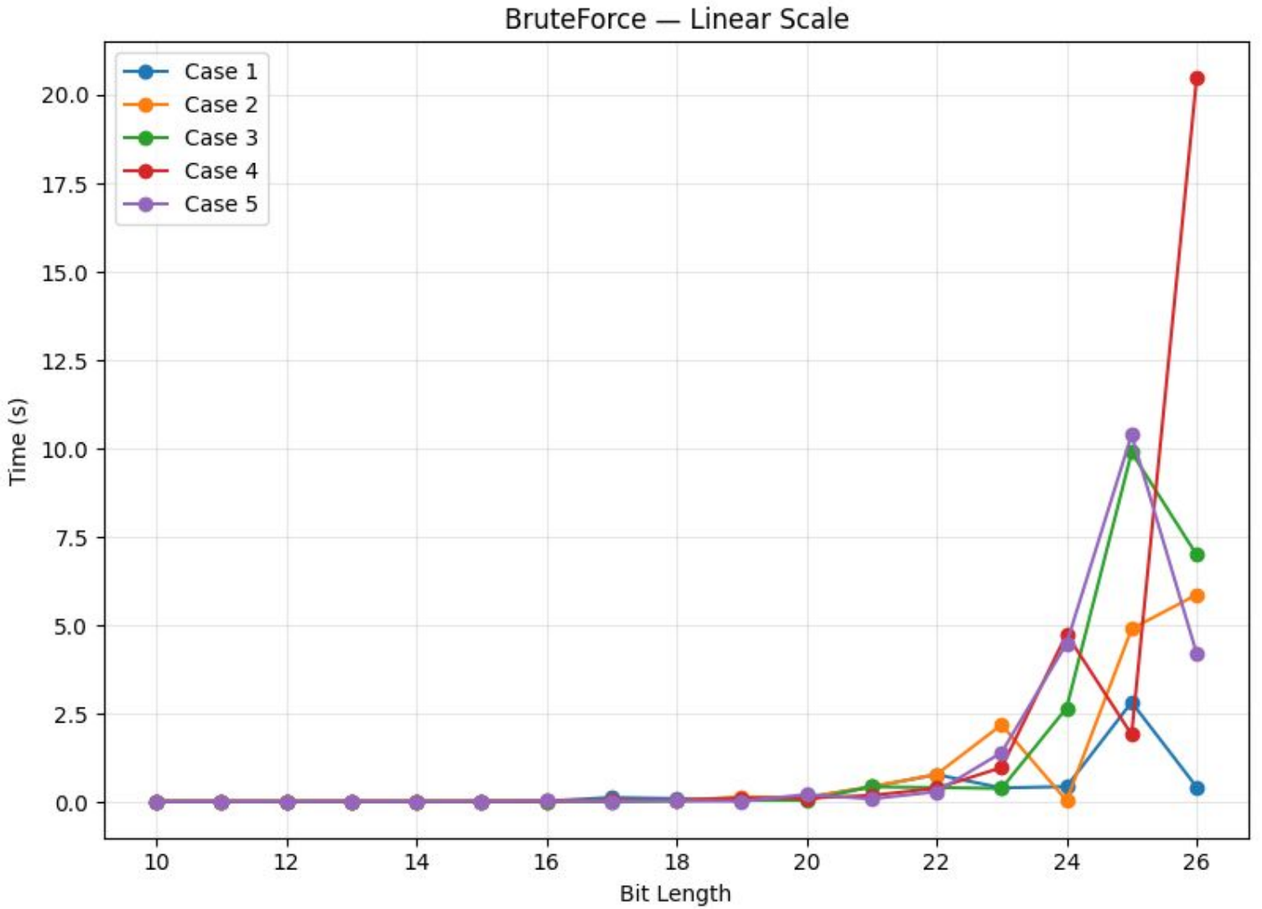
Space: *O(1)*

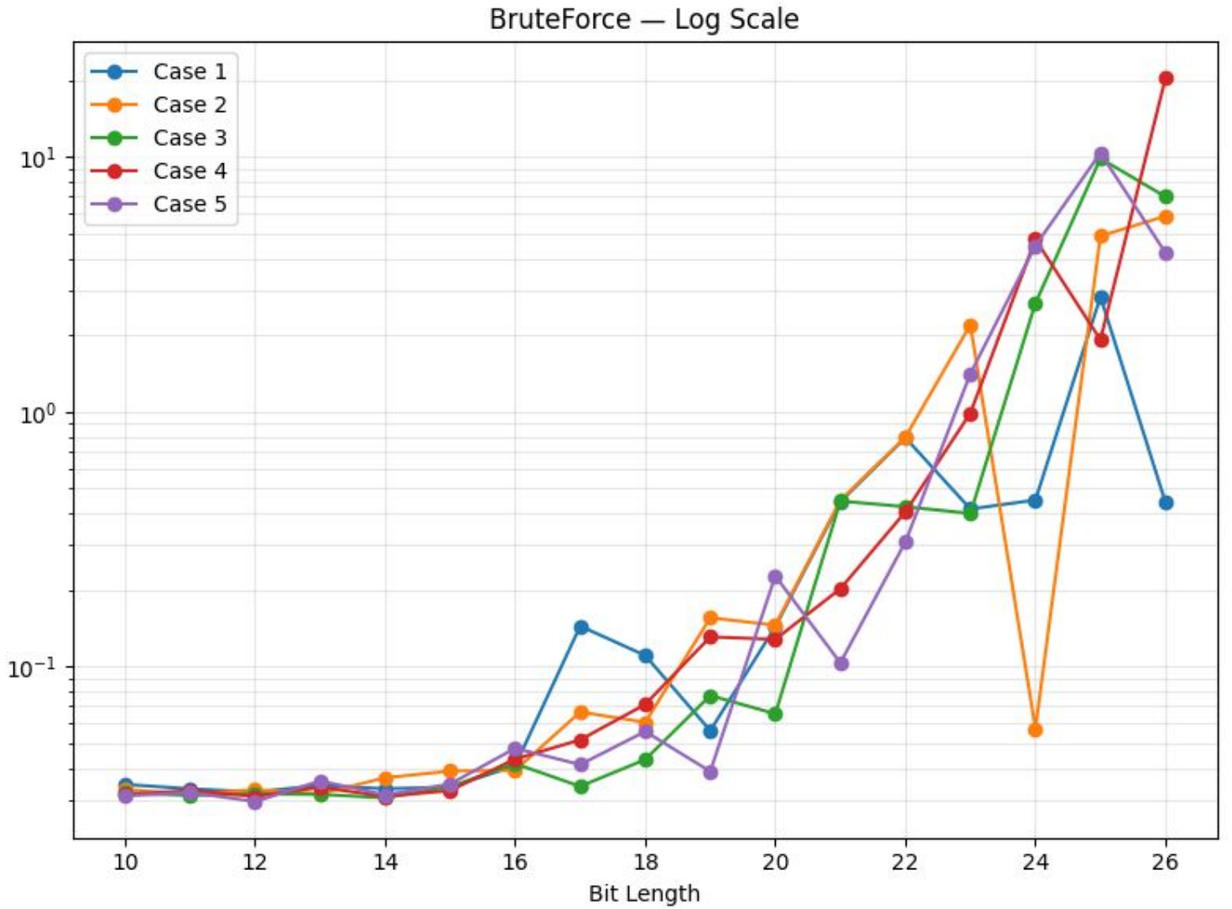
- Deterministic and requires no memory.
- Serves as a baseline for correctness validation.

Limitations & Best Use

- Extremely slow and impractical for large groups.
- Best suited for validating other algorithm results or for academic demonstration with very small curves.

Brute Force: Runtime vs. Bit Length

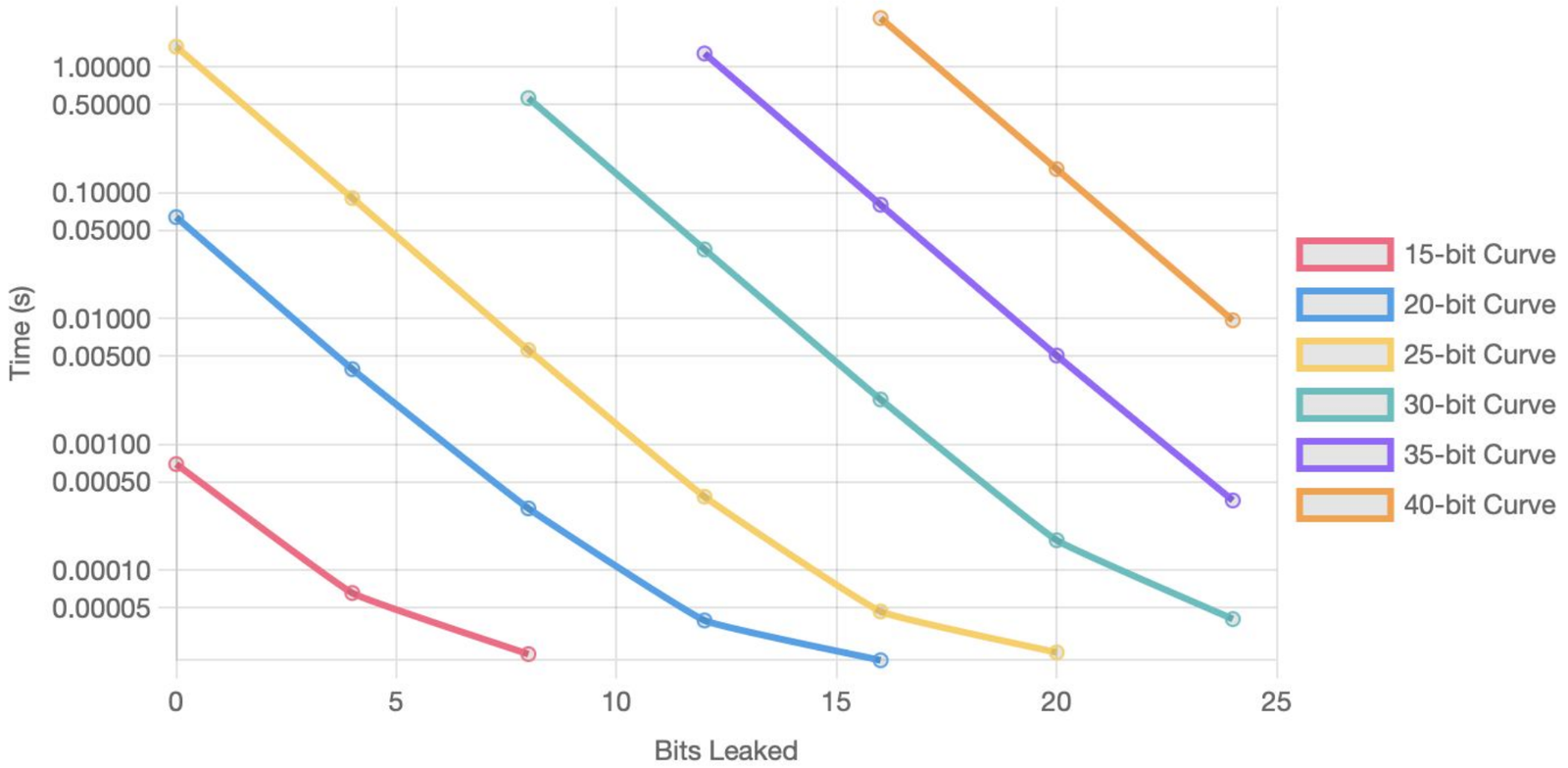




Conclusions

Beyond roughly 25 bits, the curve becomes exponentially steep, highlighting the impracticality of this method for real-world scenarios.

[BONUS] Impact on LSB Leakage



X: Bits Leaked | Y: Time (s)

Conclusions

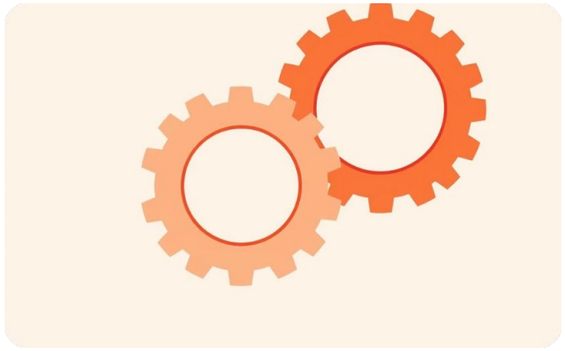
If w bits are known, the search space reduces from n to n/2^w, providing a linear speedup proportional to reduction in search space.

Baby-Step Giant-Step (BSGS): The Meet-in-the-Middle Strategy

Problem Solved: Finding the discrete logarithm 'k' where Q = kP

Description

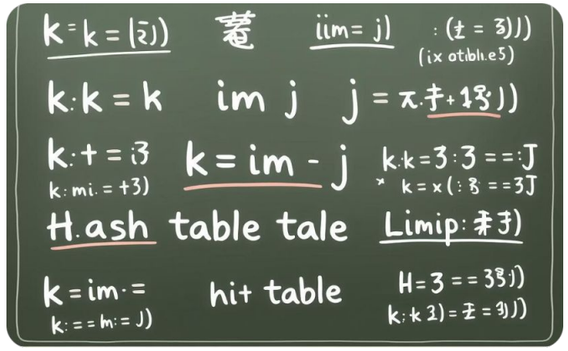
Shanks' meet-in-the-middle algorithm divides the problem into smaller, manageable parts. It writes `k = im + j`, where `m = ⌈√n⌉`.



Working Procedure

Precompute "baby steps": `jP` for `j = 0, ..., m-1` and store them in a hash table.

Compute "giant steps": iteratively calculate `Q - i(mP)` for `i = 0, ..., m-1`. Look for a match between the baby steps and giant steps in the hash table.



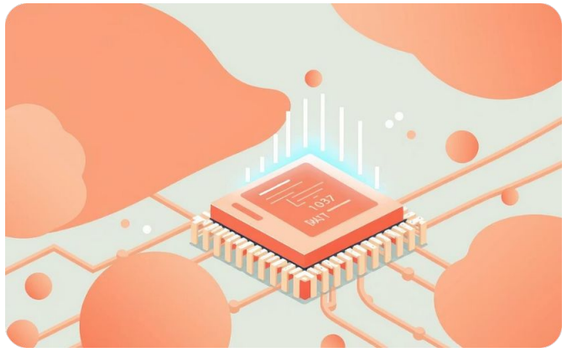
Formulas & Correctness

`k = i·m + j`

Baby steps: `Table[j] = jP`

Giant steps: `Q − i(mP)`

The algorithm is guaranteed to find a solution because every `k` within the range has a unique `(i,j)` pair.



Complexity & Features

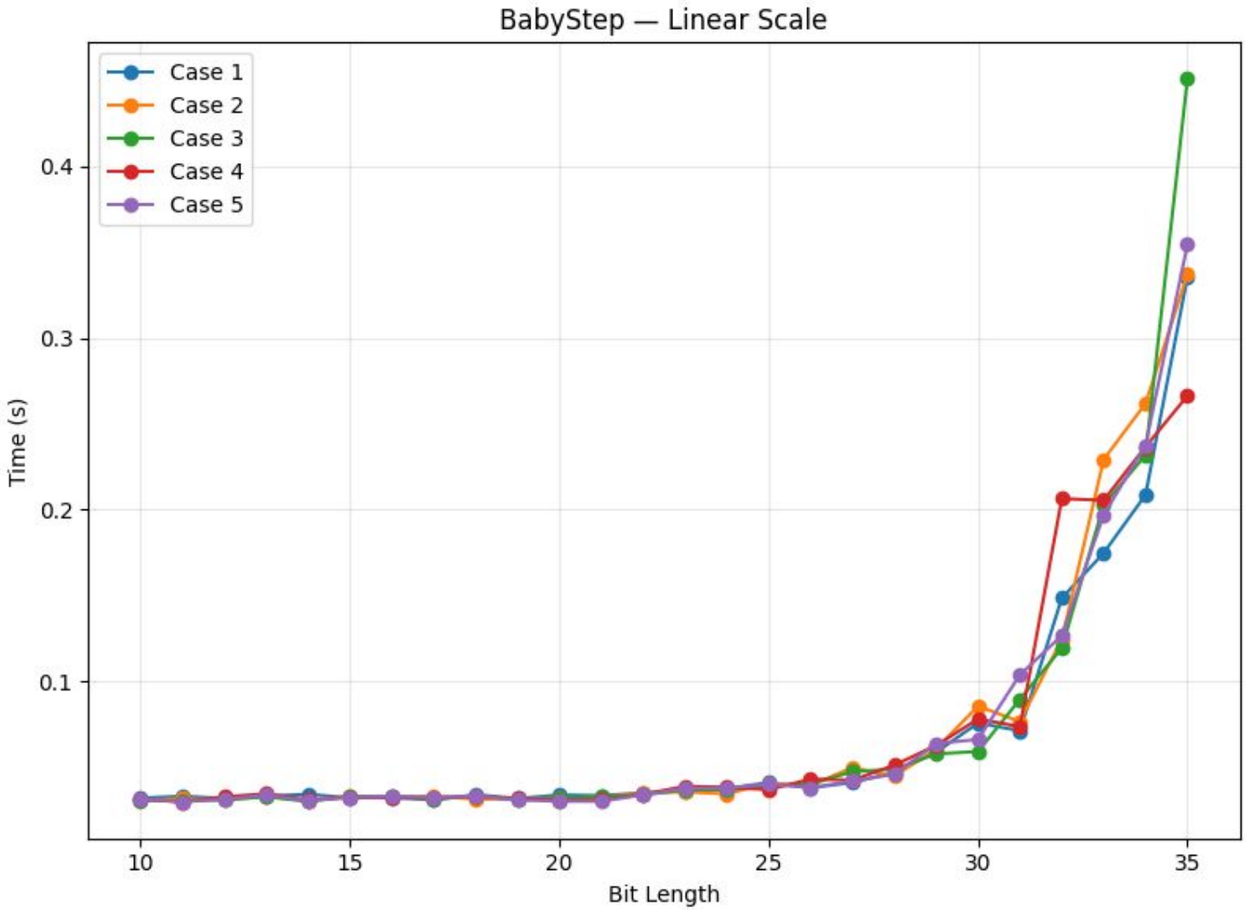
Time: *O(√n)*

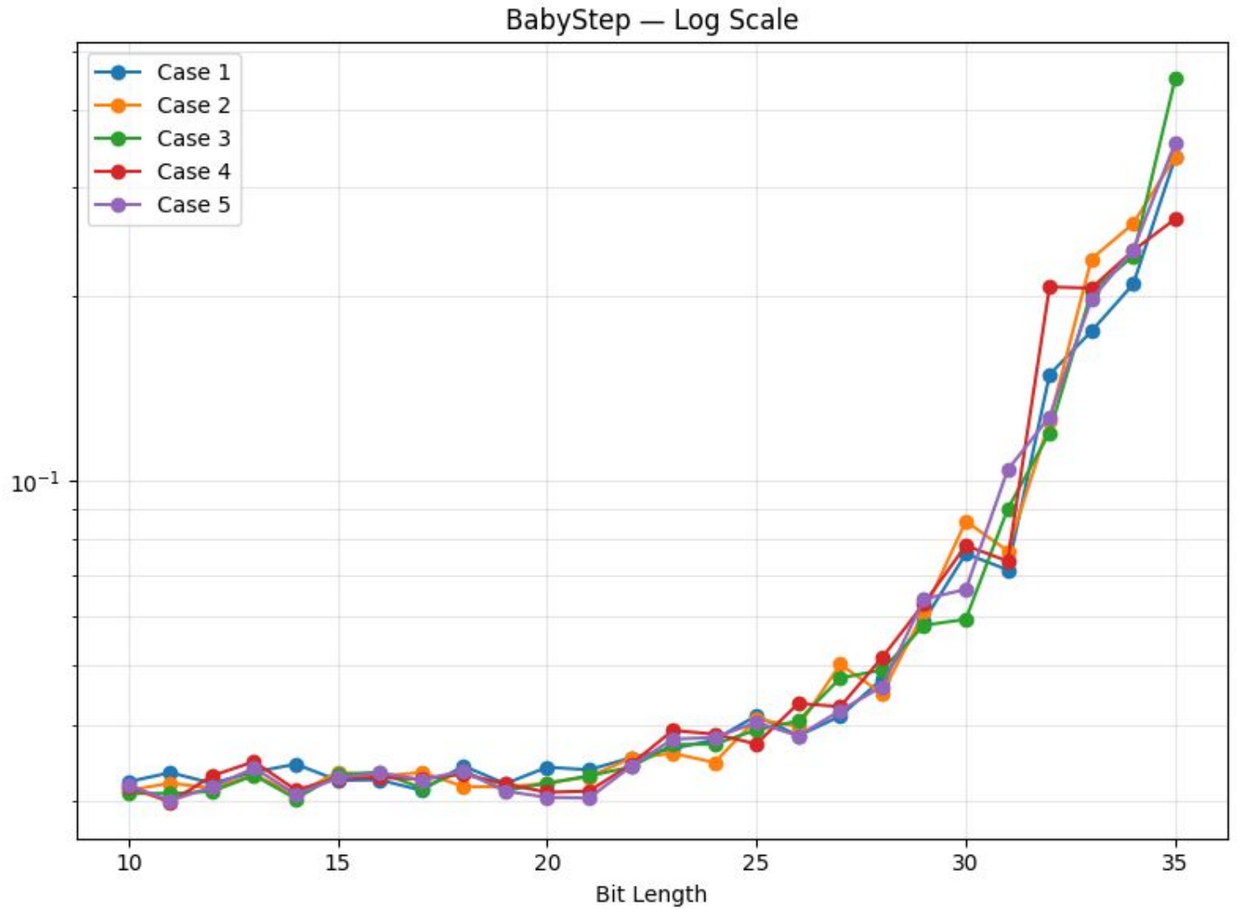
Space: *O(√n)*

- Significantly faster than brute force.
- Relies heavily on hash tables for efficient lookups.

Memory usage is its primary limitation, especially for very large `n`.

Baby-Step Giant-Step: Runtime vs. Bit Length

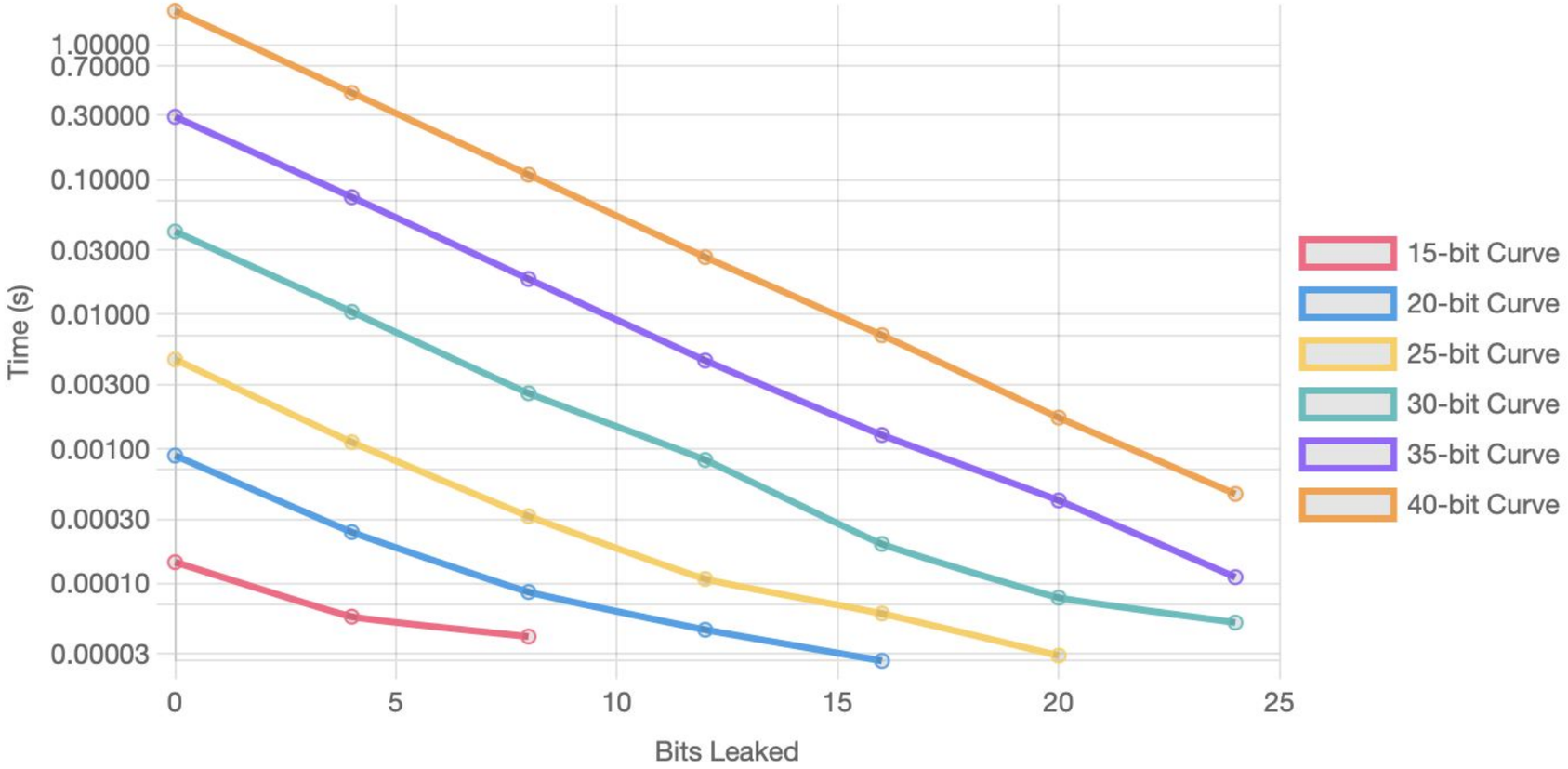




Conclusions

- The BSGS algorithm demonstrates a significantly improved runtime compared to brute force, following an $O(\sqrt{n})$ growth curve.
- While runtime is faster, the space complexity also grows as $O(\sqrt{n})$ due to the need for storing baby steps in a hash table, becoming a limiting factor for extremely large bit lengths.

[BONUS] Impact on LSB Leakage



X: Bits Leaked | Y: Time (s)

Conclusions

Both time and space complexities are reduced by a factor of approximately 2^(b/2) compared to the standard O(root n) bounds.

Pollard’s Rho: The Collision-Finding Trail

Problem Solved: Efficiently finding 'k' in Q = kP with minimal memory

Description & Working

Pollard’s Rho algorithm uses a random walk on the elliptic curve group, generating a sequence of points $X_i = a_iP + b_iQ$.

It employs partition-based update rules to guide this walk, searching for a collision where $X_i = X_j$.

Collision & Formulas

A collision $X_i = X_j$ leads to a linear congruence:

$$k \ (b_j - b_i) \equiv (a_i - a_j) \pmod{n}$$

Floyd’s cycle-finding algorithm (tortoise and hare) is typically used to efficiently detect these collisions.

Correctness & Restart

The algorithm guarantees a solution if $\gcd(b_j - b_i, n) = 1$.

If the $\gcd$ is not 1, the algorithm restarts with different initial parameters, ensuring eventual success.

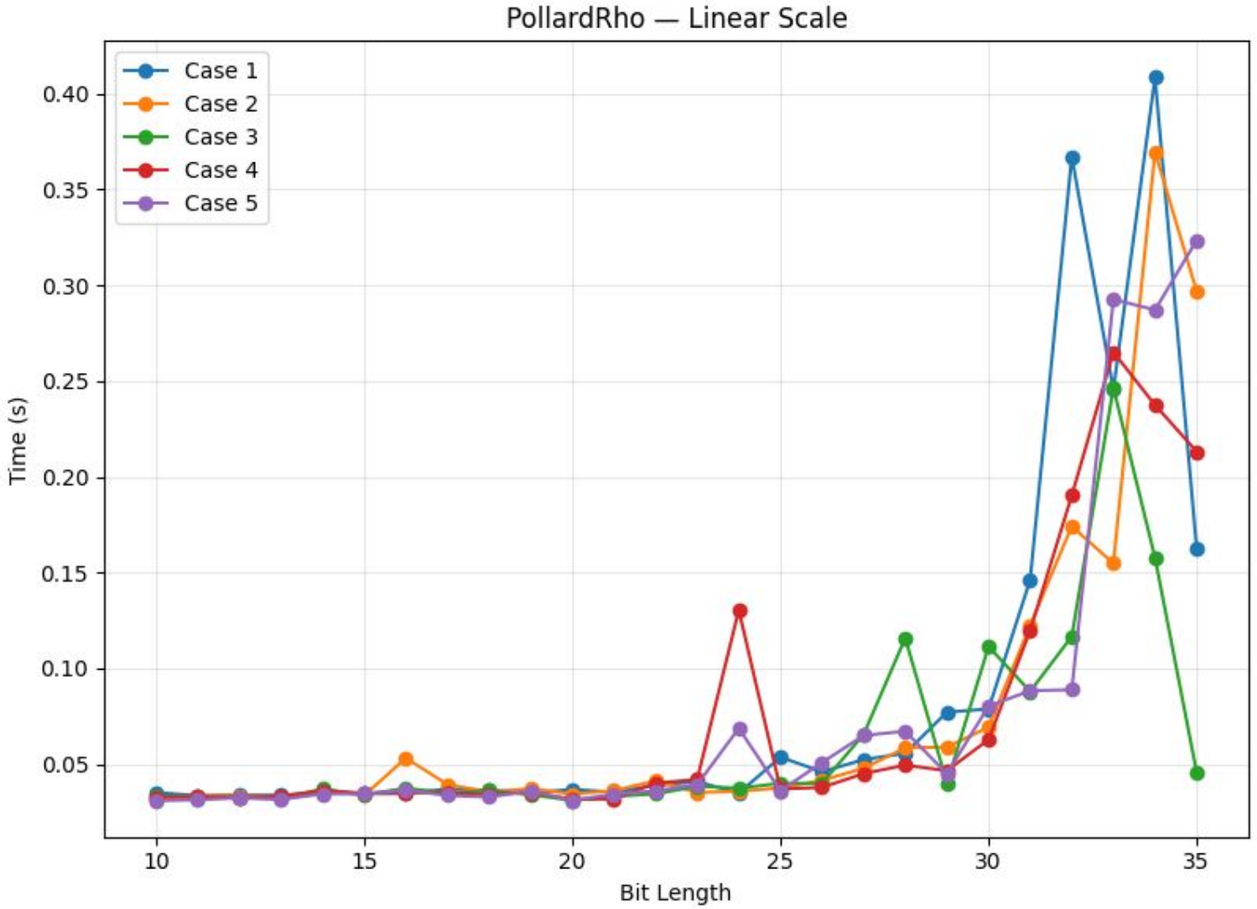
Complexity & Strengths

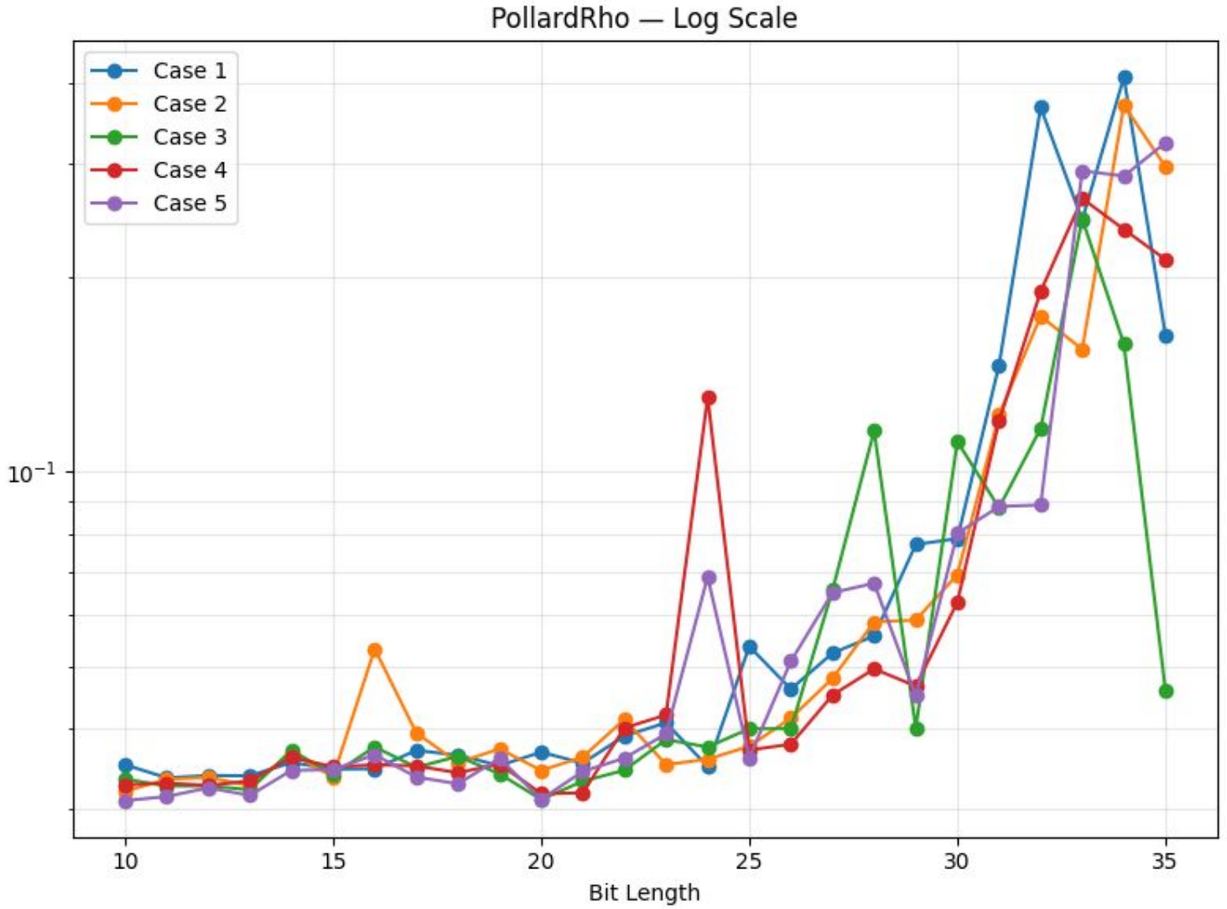
Expected Time: *O(√n)*

Space: *O(1)*

- Remarkably memory-efficient, requiring minimal storage.
- Considered the best general-purpose classical algorithm for ECDLP due to its low memory footprint.

Pollard’s Rho: Runtime vs. Bit Length

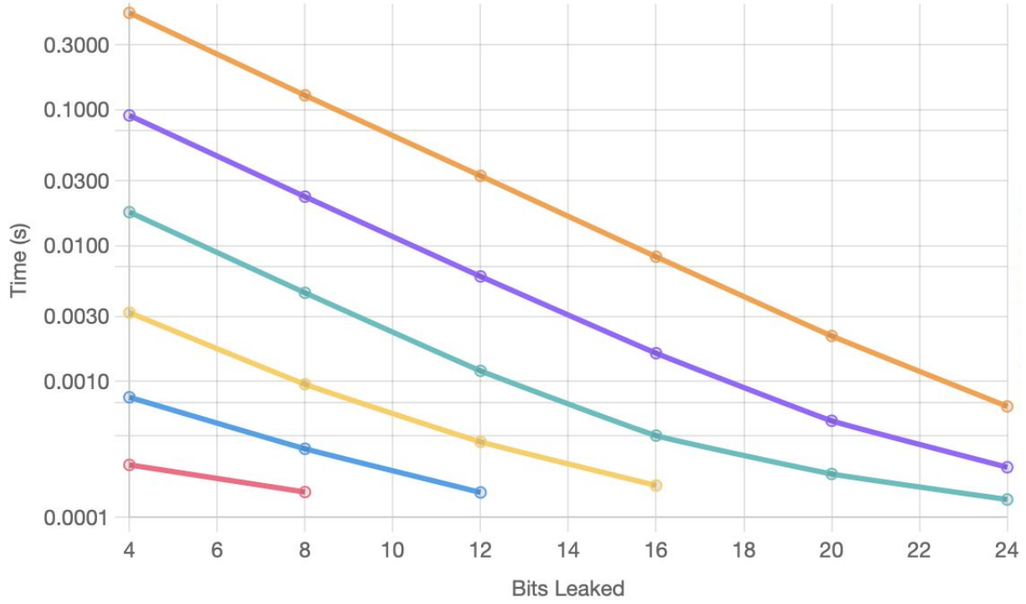




Conclusions

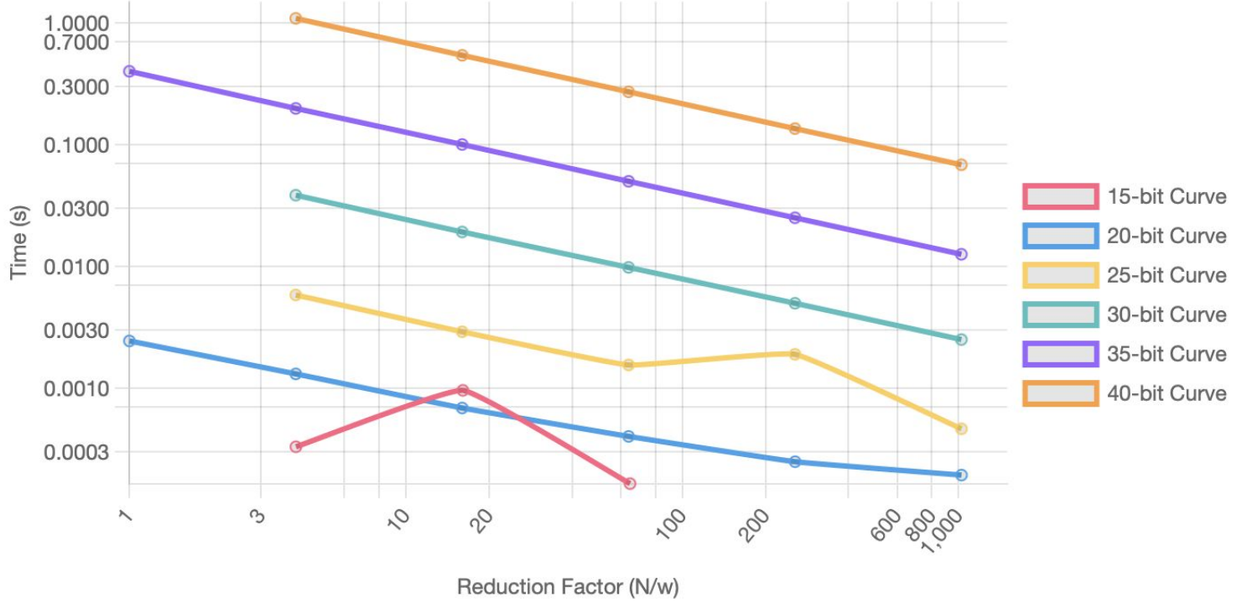
- Pollard’s Rho exhibits a runtime profile similar to BSGS, reflecting its $O(\sqrt{n})$ expected time complexity.
- A key insight is its extremely low memory usage ($O(1)$), making it the most scalable classical algorithm for ECDLP, especially when memory is a constraint.
- Randomness inherent in the algorithm can introduce variance in actual runtimes, though the expected performance remains consistent with $\sqrt{n}$.

[BONUS] Impact on LSB Leakage



X: Bits Leaked | Y: Time (s)

Interval Reduction



X: Reduction Factor (N/Width) | Y: Time (s)

Conclusions

Both time and space complexities are reduced by a factor of approximately 2^(b/2) compared to the standard O(root n) bounds.

Pohlig–Hellman: Exploiting Smooth Group Orders

Problem Solved: Accelerating ECDLP when the group order 'n' has small prime factors

Description

This algorithm is highly optimized for elliptic curves where the order of the group, n, is a "smooth" number (i.e., it has only small prime factors).
It factorizes $n = \prod p_i^{e_i}$, where p_i are prime factors.

Working Procedure

The main ECDLP is reduced into several smaller subproblems, one for each prime power factor $p_i^{e_i}$.

- Each subproblem is solved independently (e.g., using Pollard's Rho or BSGS).
- The results from the subproblems are then combined using the Chinese Remainder Theorem (CRT) to obtain the final solution.

Formulas & Correctness

Subproblem: $Q_i = (n/p_i^{e_i})Q$

The algorithm relies on knowing the full factorization of n and applying the Chinese Remainder Theorem for reconstruction.

Complexity & Notes

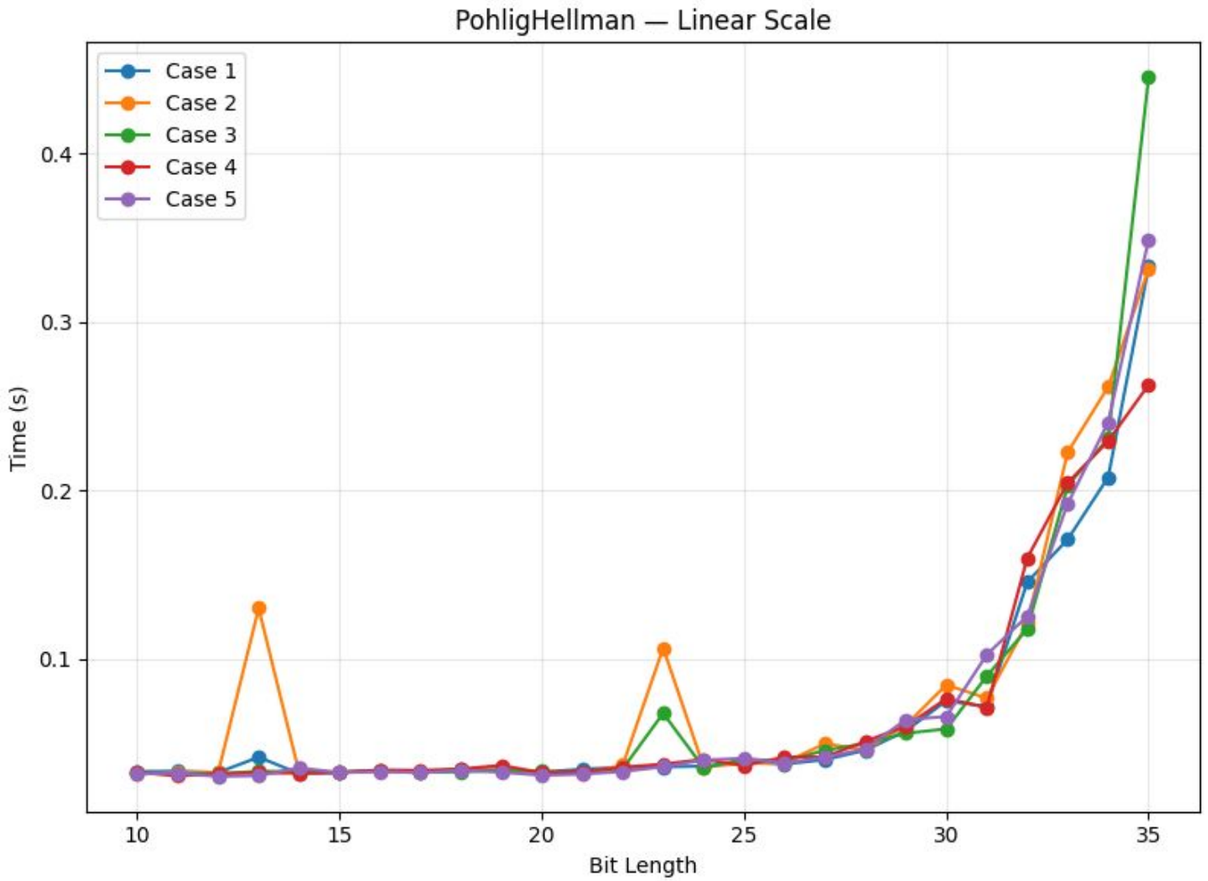
Time: $O(\Sigma \sqrt{p_i})$

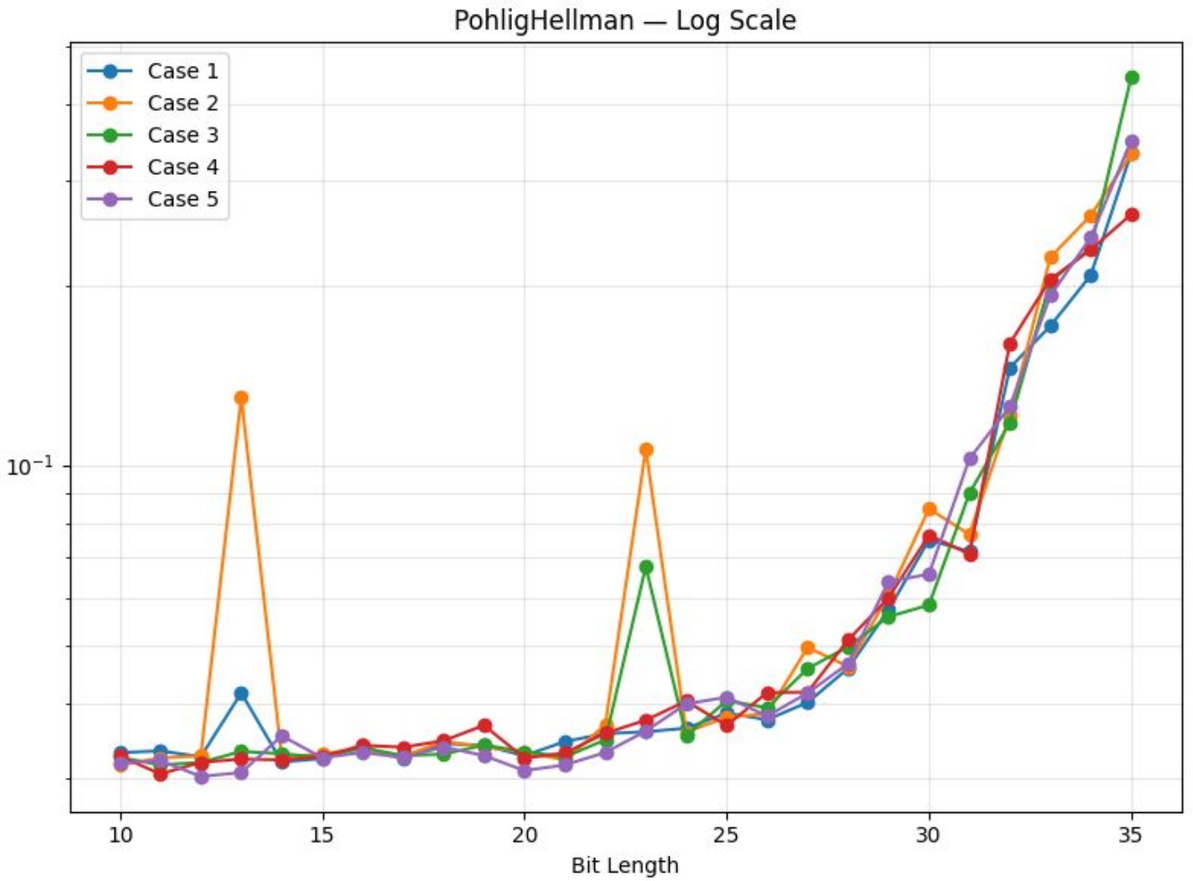
Space: $O(1)$

Extremely efficient when n has only small prime factors.

- Highlights the importance of choosing elliptic curves with large prime factors in their group order to ensure cryptographic strength.

Pohlig–Hellman: Runtime vs. Bit Length and Smoothness



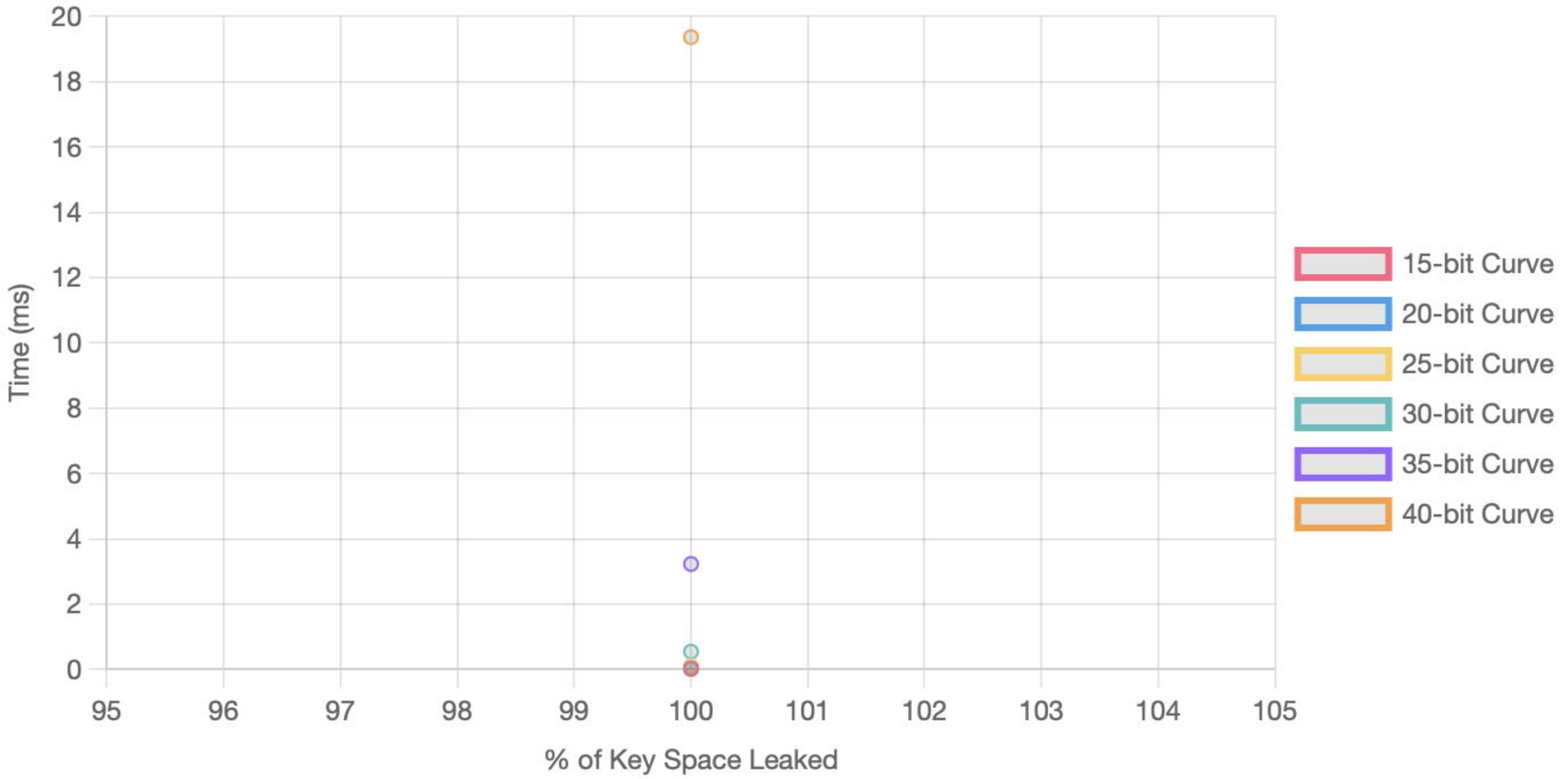


Conclusions

The Pohlig–Hellman algorithm's performance is heavily influenced by the size of the largest prime factor of the group order, showing rapid acceleration for smooth orders.

- The graph illustrates a sharp speed-up when the largest prime factor of the group order is small, appearing almost flat in those regions.
- Runtime complexity is dictated by the largest prime factor, not the overall group order, showcasing its strength on "smooth" curves and its weakness on "rough" curves.
- This behavior underscores the critical need for cryptographic curve selection to ensure the group order has a sufficiently large prime factor to resist Pohlig-Hellman attacks.

[BONUS] Residual Leakage



X: % of Key Space Leaked | Y: Time (ms) - Shows impact of leaking largest factor

Conclusions

Leaking the largest prime factor of n is catastrophic, as it reduces the security of the discrete logarithm to that of the second largest factor.

Las Vegas Algorithms: Probabilistic Efficiency

Problem Solved: Finding 'k' with guaranteed correctness but variable runtime

Description

Las Vegas algorithms are a class of probabilistic algorithms that always produce a correct result, but their runtime is variable due to the use of randomness.
This category includes randomized versions of algorithms like Pollard's Rho and the Kangaroo algorithm.

Working Principle

- They often involve multiple randomized restarts to explore different search paths.
- Techniques like random partitioning are used to introduce variability and improve the chance of early collisions.

Key Concepts

Randomization strategically shifts the distribution of collisions, potentially leading to faster discovery.
Some variants, as noted in advanced reports, utilize summation polynomial-based approaches for further optimization.

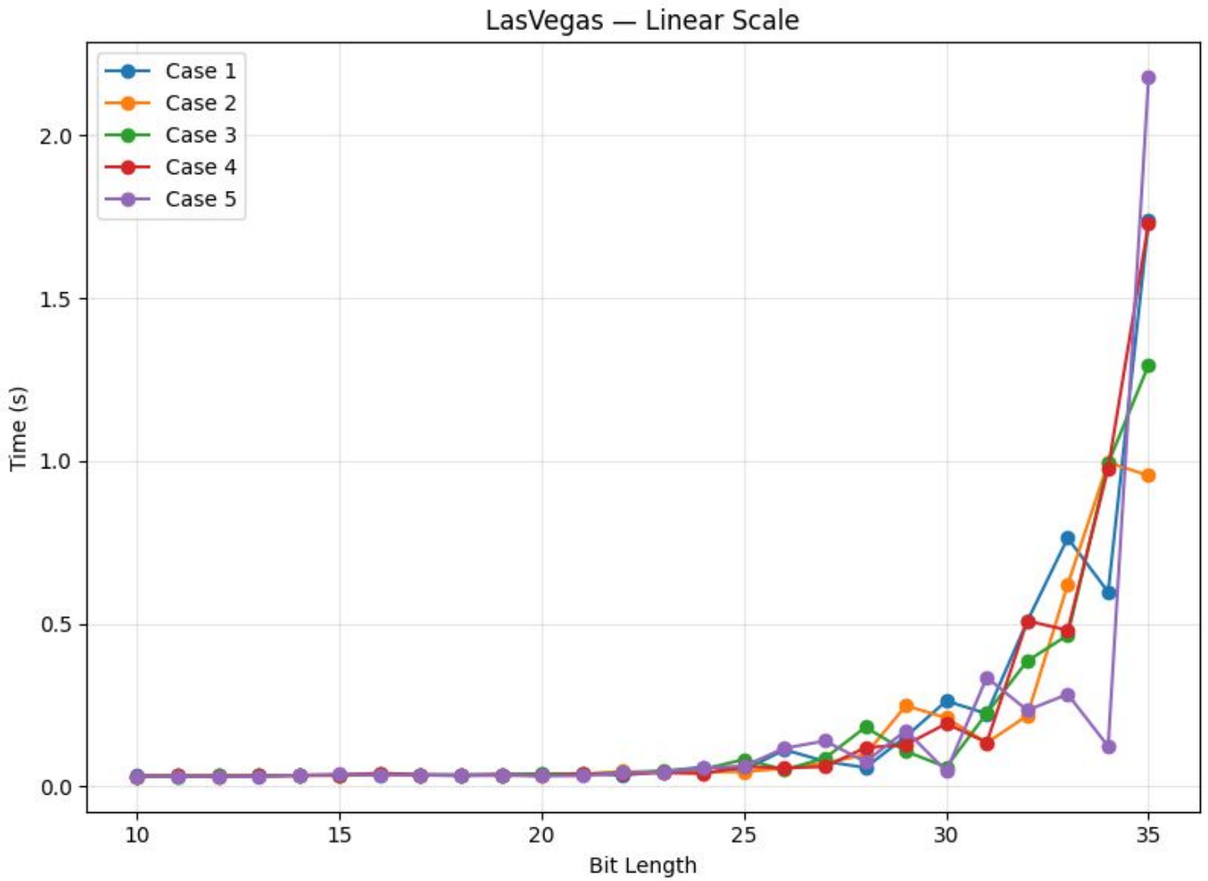
Complexity & Notes

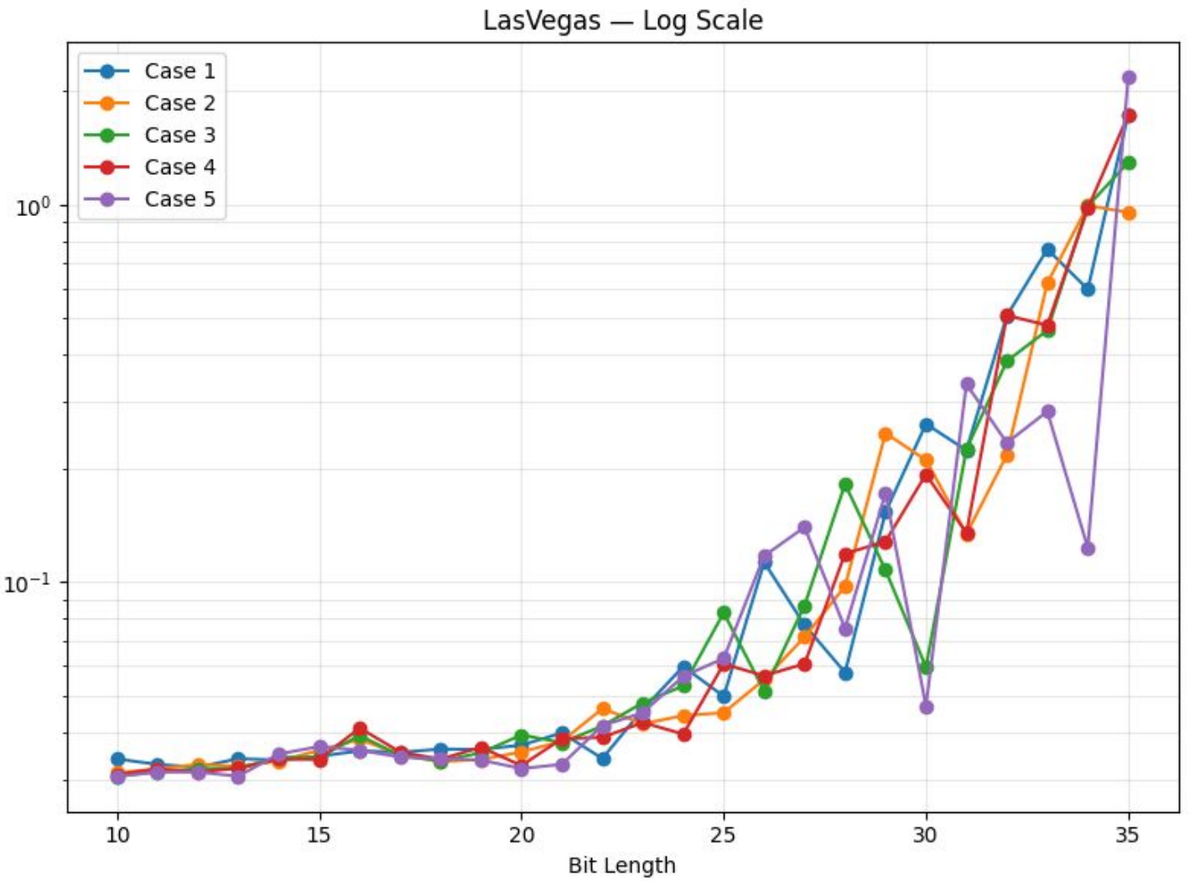
Expected Time: $O(\sqrt{n})$

Worst-Case: Unbounded, though practically rare.

Runtime variance can be high, but they are particularly useful when external leakage or hints about `k` are available, allowing for more targeted searches.

Las Vegas: Runtime vs. Bit Length and Smoothness

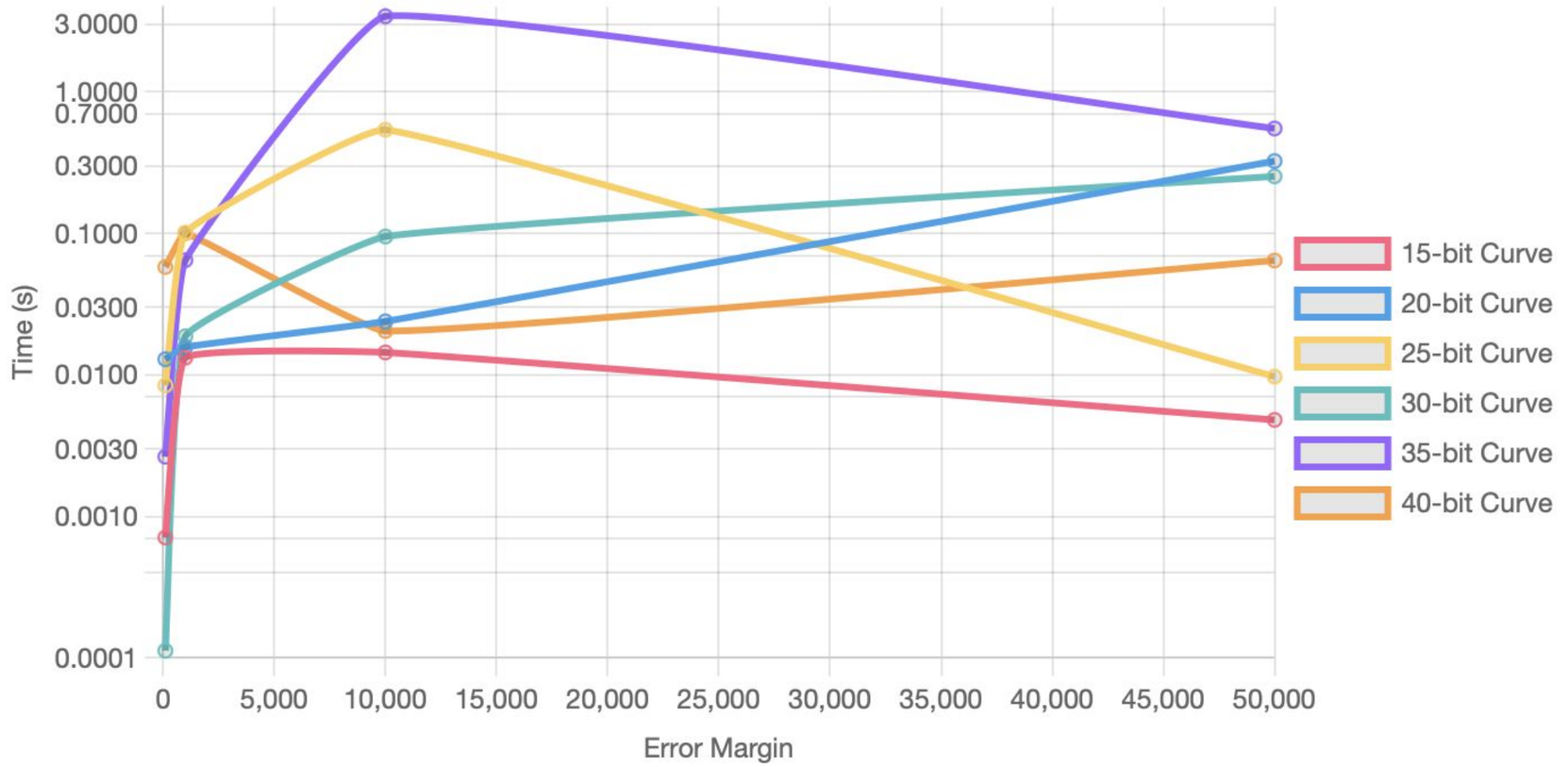




Conclusion

It shows that it exhibits an expected o(root n) runtime complexity, consistent with Pollard's Rho.

[BONUS] Approximate Key Knowledge



X: Error Margin (+/-) | Y: Time (s)

Implementation Details: Technical Deep Dive

Elliptic Curve Arithmetic

Curve Form: Implemented using the short Weierstrass equation: $y^2 = x^3 + ax + b \pmod p$.

Core Operations: Manual implementation of point addition, point doubling, and point negation operations on the elliptic curve points.

Modular Inverse: Utilized the Extended Euclidean Algorithm to compute modular inverses, essential for division in finite fields.

Coordinate Systems: For performance, projective coordinates were selectively employed to minimize expensive modular inverse calculations.

Data Structures & State Management

BSGS Storage: Dictionaries (hash maps) were used to store the "baby steps" for efficient lookup, mapping points to their corresponding discrete logarithms.

Point Representation: Elliptic curve points were represented as tuples (x, y).

Pollard's Rho: Employed a partition-based state machine to manage the random walk iterations, categorizing points to ensure progress and detect collisions.

Las Vegas Variants: Utilized flags for state transitions and random restarts to enhance robustness and explore new segments of the search space upon stagnation.

Collision Handling

Pollard's Rho: Implemented Floyd's cycle-detection algorithm (tortoise and hare) to efficiently find collisions within the random walk.

Collision Equation: Once a collision is detected, the discrete logarithm k is derived by solving the linear congruence: $k(b_2 - b_1) \equiv (a_1 - a_2) \pmod n$, where n is the order of the generator point.

Challenges Faced

Point-at-Infinity: Correctly handling the point at infinity, which acts as the additive identity element on the curve.

Random Walk Loops: Ensuring that random walks in Pollard's Rho and its variants do not prematurely enter short, unproductive cycles.

Memory Explosion: Managing the memory requirements for BSGS, which can become prohibitive for larger group orders.

Modular Arithmetic Errors: Meticulously debugging potential errors in modular arithmetic operations, which are foundational to correct curve computations.

Experimental Methodology: Benchmarking & Validation

To rigorously evaluate the implemented ECDLP algorithms, a comprehensive experimental methodology was designed, focusing on systematic benchmarking, parameter sweeps, and robust validation.

Benchmarking Process

Multiple Runs: Each algorithm was executed 5 to 20 times per specific bit length of the elliptic curve to ensure statistical significance and mitigate effects of system variability.

Wall-Clock Runtime: Averaged wall-clock runtime measurements were collected to provide a realistic assessment of computational performance.

Memory Footprint: Memory consumption was recorded by tracking table sizes in BSGS and storage depth for other algorithms, highlighting memory bottlenecks.

Collision & Restart Logging: For probabilistic algorithms, collision counts (Pollard's Rho) and restart counts (Las Vegas variants) were meticulously logged to understand their operational dynamics.

Parameter Sweeps

Curve Sizes: Elliptic curves ranging from 20-bit to 50-bit prime fields were tested to observe scaling behavior across different security levels.

Group Order Characteristics: Experiments included both "smooth" and "non-smooth" group orders to assess the effectiveness of algorithms like Pohlig–Hellman.

Leakage Scenarios: Simulated scenarios included:

Known Least Significant Bits (LSB) of k.

Interval bounds on the possible values of k.

Known residues of k modulo small prime factors, specifically for Pohlig–Hellman.

Graph Generation & Validation

Visual Analysis: Data was visualized through graphs depicting runtime versus bit length, leakage impact versus speed-up factors, and comparative performance curves for all algorithms.

Verification: Every computed discrete logarithm k was rigorously verified by confirming that $kP = Q$. Brute Force was additionally used as a correctness baseline for validating other algorithm implementations.

Results Summary: Key Insights and Discoveries

The experimental results provide empirical validation of theoretical complexities and reveal critical implications for ECC security, especially concerning information leakage.

Brute Force

Exhibited clear linear growth with respect to the group order, confirming its infeasibility for cryptographic key sizes.

BSGS

Matched the expected $\sqrt{n}$ trend in runtime, but its $O(\sqrt{n})$ memory requirement quickly became the dominant bottleneck for larger curves.

Pollard's Rho

Demonstrated the same $\sqrt{n}$ runtime complexity as BSGS but with a crucial advantage of $O(1)$ memory, albeit with slight runtime variance due to its probabilistic nature.

Pohlig–Hellman

Proved exceptionally fast when the group order had small prime factors. Its performance was primarily dictated by the size of the largest prime factor of the group order.

Las Vegas Variants

While exhibiting higher variance in runtime, these probabilistic algorithms frequently outperformed deterministic Pollard's Rho on average, particularly with specific initializations or strategies.

Leakage Experiments: Security Implications

LSB Knowledge: Knowing b LSBs of k reduced the effective complexity from $\sqrt{n}$ to $\sqrt{(n / 2^b)}$, significantly accelerating attacks.

Interval Bounds: Providing interval bounds for k made algorithms like the Pohlig–Hellman with Kangaroo's algorithm extremely efficient within the bounded range.

Residue Knowledge: Knowledge of k modulo small prime factors allowed Pohlig–Hellman to solve subproblems and recombine results via the Chinese Remainder Theorem, bypassing full DLP.

Main Takeaway

> Even seemingly minor leakage of information about the discrete logarithm k severely weakens the hardness of ECDLP, demonstrating the extreme sensitivity of ECC security to implementation and side-channel vulnerabilities.

Challenges Faced During Implementation

Developing robust and correct implementations of ECDLP-solving algorithms from scratch presented several intricate challenges, demanding careful attention to detail and rigorous debugging.

1

Full EC Arithmetic from Scratch

Implementing all elliptic curve arithmetic operations (point addition, doubling, negation, scalar multiplication) accurately was complex, especially ensuring correctness over finite fields and handling edge cases.

2

Debugging Coordinate System Errors

Distinguishing and debugging errors between affine coordinate formulas and more efficient projective coordinate implementations proved challenging, often requiring careful cross-verification.

3

Pollard's Rho: Short Cycles

Ensuring the random walk in Pollard's Rho avoided prematurely entering short, unproductive cycles, which could lead to incorrect results or infinite loops, required careful design of the partition function.

4

Collision Equation Solvability

Guaranteeing the solvability of the collision equation $k(b_2 - b_1) \equiv (a_1 - a_2) \pmod{n}$ and handling cases where $\gcd(b_2 - b_1, n) \neq 1$ was crucial for correctness.

5

BSGS Memory Requirements

Managing the significant memory demands of the Baby-Step Giant-Step algorithm for larger group orders posed a practical limitation and a design challenge.

6

Extensive Experimentation & Logging

Organizing and logging a large number of experimental runs and their associated metrics (runtime, memory, collision counts) for different parameters required robust infrastructure.

7

Las Vegas Restarts

Implementing random restarts for Las Vegas variants without introducing infinite loops or biases, while ensuring proper exploration of the search space, was a delicate balance.

8

Kangaroo Stride Points

Safely precomputing stride points for the bounded-interval Kangaroo algorithm, particularly ensuring their randomness and distribution, was a specific challenge for that variant.

Experimental Setup: Rigor and Reproducibility

Our experiments were conducted under controlled conditions to ensure accurate and reproducible results, providing a robust foundation for our empirical analysis.

1

Hardware Environment

CPU: M1 Pro 8 Core

RAM: 16 GB for memory-intensive tasks

2

Software Stack

Programming Language: Python 3.x

EC Library: Custom-built from scratch (no external crypto dependencies)

Visualization: Matplotlib for data plotting

Finite Fields: Small prime fields (F_p)

Group Orders: Varied from 20 to 240 bits

Properties: Both smooth and non-smooth orders

Leakage: LSB bits (0–20), interval sizes

Collected Metrics

Performance: Wall-clock time, memory usage

Algorithm Specific: Collision counts (Rho), retries (Las Vegas)

Reliability: Success probability across runs

3

Curve Parameters

4

Rigorous Testing Environment for ECC Benchmarks

This presentation details the experimental setup and methodology used for benchmarking Elliptic Curve Cryptography (ECC) Discrete Logarithm (DL) algorithms. Our goal is to provide a comprehensive analysis of algorithm performance under controlled and reproducible conditions.

Hardware Consistency

- Standard workstation (quad-core or higher CPU)
- 8–16 GB RAM
- All algorithms executed on the same machine to guarantee fair comparisons and eliminate environmental variables.

Software Foundation

- Python 3.x with a custom, from-scratch elliptic curve arithmetic implementation (no external crypto libraries used).
- Matplotlib for detailed graph generation.

Essential utilities: `subprocess`, `time`, `pathlib`, `re`.

Optional C++ acceleration for scalar multiplication via `ctypes`.

ECC Curve Parameters & Metrics Recorded

Our benchmarks utilize specific curve parameters to explore various scenarios, and a comprehensive set of metrics are recorded to ensure thorough analysis.

Curve Specifications

Finite fields: Prime fields **Fp** with $p \equiv 11 \pmod{12}$.

Group orders: **Prime subgroups q = (p+1)/12**.

Bit lengths tested: Configurable from **10 to 60 bits**.

Test cases: **5 distinct cases per bit length** (e.g., `case_1.txt` to `case_5.txt`).

- Curve form: Short Weierstrass equation $y^2 = x^3 + ax + b$.
- Includes both generic curves and special cases (smooth orders, anomalous curves).

Leakage Scenarios & Metrics

LSB leakage: Exploring scenarios where 0–20 least significant bits are known.

Interval search: Utilizing Pollard's Kangaroo algorithm for bounded intervals.

Smooth order: Applying Pohlig-Hellman to factorizable orders.

Wall-clock runtime per algorithm, per test case.

Average runtime across the 5 cases for each bit length.

Success rate: (passed_cases / total_cases).

Memory usage: Implicitly tracked through BSGS hash table size.

Attempts count: For probabilistic algorithms like Pollard Rho and Las Vegas.

Verification: All computed solutions are validated using $Q = d \cdot G$.

Environmental Guarantees

To ensure validity and consistency, all test cases are pre-generated with known answers. A strict 60-second timeout is enforced per test case. Reproducibility is guaranteed by reusing the same test cases across all runs, and outputs are verified via point multiplication. Smart limits are applied, skipping BruteForce beyond 26 bits and BabyStep beyond 50 bits to manage computational resources effectively.

Implementation Details: ECC Arithmetic and Data Structures

Our custom implementation of Elliptic Curve arithmetic and carefully selected data structures are fundamental to the benchmark's accuracy and performance.

Elliptic Curve Arithmetic

Implemented in `utils/ecc_utils.py`.

- Supports $y^2 \equiv x^3 + ax + b \pmod{p}$ Short Weierstrass form.

`EllipticCurve` class with methods for point addition (`add`), scalar multiplication (`scalar_multiply` using double-and-add), point negation (`negate`), and point validation (`is_on_curve`).
Point at infinity represented as `None`.

Modular inverse computed using Fermat's Little Theorem (`pow(a, p-2, p)`).

Baby-Step Giant-Step (BSGS)

Utilizes a Python `dict` to map `point → baby-step index`.

- Space complexity: $O(\sqrt{n})$ entries.
- Average case lookup time: $O(1)$.

Pollard's Rho

Employs a partition table of 32 random jump points `R_i = u_i·G + v_i·Q`.

Maintains state `(X, A, B)` where `X = A·G + B·Q`.

- Uses Floyd's cycle detection (tortoise vs. hare) for efficiency.

Pohlig-Hellman & Las Vegas

Pohlig-Hellman: Factors 'n' into prime powers, solves sub-problems in subgroups, and combines results using the Chinese Remainder Theorem.

Las Vegas: Employs a random point sampling strategy, particularly effective for special curves, with fallback restarts.

Optimizations & Collision Detection

C++ acceleration (`utils/cpp/ecc_fast.so`) offloads scalar multiplication for 10-100x speedup via `ctypes`, with graceful Python fallback. Dynamic configuration adjusts Pollard Rho parameters based on curve size. Smart limits prevent exhaustive computation for large bit lengths. Pollard Rho's collision detection uses Floyd's cycle finding ($O(1)$ space), resolving $A_1 \cdot G + B_1 \cdot Q = A_2 \cdot G + B_2 \cdot Q$ to find d.

Experimental Methodology: Benchmarking & Analysis

Our methodology ensures repeatable and accurate results, covering test case generation, comprehensive benchmarking, and robust data collection.

1 Test Case Generation

Utilizes `generate_test_cases.py` with start and end bit lengths

Primes 'p' are selected such that $p \equiv 11 \pmod{12}$, and 'q' is a prime group order $q = (p+1)/12$.

- Five unique test cases are generated per bit length, each with a known solution, ensuring thorough coverage.
- Generated curves are of the form $y^2 = x^3 + ax + b$ (default $a=0$).

A generator point G of exact prime order q is verified ($q \cdot G = O$), and $Q = d \cdot G$ is computed for a random secret 'd'.

Test cases and answers are saved in `test_cases/bit/case_*.txt` and `answer_*.txt` respectively.

2 Benchmarking Process

The `run_comparison.py` script manages the overall benchmarking.

Each algorithm is run per test case with a 60-second timeout.

- Output parsing captures success/failure, and records elapsed time.
- For each bit length, all 5 cases are run, and average runtime is computed from successful cases.
- Results include individual case times, average time, and success rate.
- Smart limits prevent time-consuming operations, e.g., BruteForce is skipped beyond 26 bits, BabyStep beyond 50 bits.
- Attempts for probabilistic algorithms (Pollard Rho, Las Vegas) are tracked.

3 Data Collection & Verification

Wall-clock runtime measured with `time.perf_counter()`.

- Success detected by parsing stdout for keywords ("Solution", "PASSED", "d =").
- Attempt counts parsed via regex for probabilistic algorithms.
- Memory usage of BSGS is implicitly tracked, scaling with $\sqrt{n}$.

Primary verification: Each algorithm validates $Q = d \cdot G$ before reporting success.

Secondary verification: Cross-check against `answer_*.txt` files.

- Ground truth for smaller curves (<20 bits) is established using BruteForce.
- Timeouts or incorrect answers are counted as failures.

4 Graph Generation & Parameter Sweeps

Matplotlib is used to generate various graphs, saved to the `graphs/` folder, including `comparison_*.png` (linear + log scales, all algorithms), `individual_*.png` (one subplot per algorithm), `*_cases_*.png` (per-algorithm case breakdown), and `case_*_algo_*.png` (all algorithms for a specific case). Bit lengths are configurable, with a default of 10-30 bits and a maximum of 60 bits tested. Leakage experiments, managed by `run_bonus_scenarios.py`, test LSB leakage (5-20 bits), interval search (Kangaroo with widths 10^3 to 10^6), and Pohlig-Hellman on smooth order curves.

How to Run the Code: A Practical Guide

This section provides instructions for navigating the project repository and executing the implemented ECDLP-solving algorithms and benchmarking scripts.

Repository Structure

The project is organized into logical directories to separate concerns and facilitate navigation:

/algorithms/: Contains the individual Python scripts for each ECDLP-solving algorithm (Brute Force, BSGS, Pollard's Rho, Pohlig–Hellman, Las Vegas variants).
/utils/: Houses utility functions, including modular arithmetic operations (e.g., modular inverse, exponentiation) and core elliptic curve point operations (addition, doubling, negation).
/utils/cpp/: Optimized codes for modular inverse, point addition, scaler multiplication implemented in C++

Requirements

Python Version: Python 3.x is required for all scripts.

Dependencies: The implementation relies solely on standard Python libraries. No external cryptographic libraries are used to ensure the "from scratch" nature of the project.

Graphing (Optional): For visualizing the results, `matplotlib` is used. Ensure it is installed if graph generation is desired.

Run Instructions

To execute individual algorithms on predefined toy elliptic curves, navigate to the /algorithms/ directory and run the respective Python scripts:

```
python <algoname>/main_optimized.py path/to/testcase
```

Benchmark Runner

For comprehensive benchmarking across various curve parameters and to reproduce the experimental results, execute the main benchmark runner script:

```
python run_comparison.py
python leak_analysis.py
```

Key Takeaways

- Generic ECDLP algorithms (BSGS, Pollard's Rho) match the expected $O(\sqrt{n})$ behavior on toy curves.
- Memory is the main limitation for BSGS; Pollard's Rho is more memory-friendly.
- Pohlig–Hellman highlights how *curve parameter choices* directly affect security.
- Interval-based / probabilistic algorithms (Kangaroo, Las Vegas variants) become very powerful when partial information is available.
- Partial key leakage can turn a "theoretically secure" instance into one solvable much faster.

Conclusion & Future Directions

What We Accomplished

- Implemented and experimentally compared multiple ECDLP-solving algorithms
- Demonstrated impact of partial information on practical attack complexity

Future Work

- Move to larger curves and more realistic security levels
- Explore parallelization and GPU / hardware acceleration
- Investigate side-channel and leakage models beyond simple bit / interval hints

Final Takeaway

Choosing secure curve parameters is not enough; implementation and leakage also matter.

References

A curated list of academic papers and resources that informed the theoretical understanding and implementation of the algorithms discussed.

- Attacks on Elliptic Curve Cryptography Discrete Logarithm Problem (EC-DLP) – IJIREEICE
- A New Method for Solving the Elliptic Curve Discrete Logarithm Problem – GCC
- A Las Vegas Algorithm to Solve the ECDLP – Indocrypt Talk
- A Review: Solving ECDLP Problem Using Pollard's Rho Algorithm – IJARCCE
- Classical Attacks on Elliptic Curve Cryptosystems – Osaka University
- Pohlig–Hellman Applied in Elliptic Curve Cryptography – RootMe
- Index Calculus — MIT 18.783 Lecture 10
- Research Status of the ECDLP — CA/Browser Forum
- ECC Attacks Repository — GitHub