ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Department of Electrical, Electronic, and Information Engineering
"Guglielmo Marconi"

Software Development M

Cryptographic Algorithms: AES, RSA, and Perfect Hash Functions

An Experimental Study of Symmetric and Asymmetric Encryption with Advanced Hash Function Techniques

Juraj Sýkora

January, 2026

Academic Year 2025/2026

0.1 Abstract

The aim of this work is to get to know and deeply understand the basics of modern cryptography, as well as some techniques for the analysis of security protocols. This paper presents an implementation and experimental study of chosen cryptographic algorithms, including AES-128 (Advanced Encryption Standard), RSA (Rivest-Shamir-Adleman), and Perfect Hash Functions (PHF).

The goal of this research was to gain a deep understanding of how these algorithms work internally, their performance characteristics, and practical implementation considerations. Through building these systems from scratch in C/C++, exploring the intricate details of symmetric encryption (AES), asymmetric encryption (RSA), and advanced hashing techniques (CHD and BDZ algorithms for perfect hashing).

0.2 Introduction

This research takes an experimental approach by implementing three categories of cryptographic primitives from scratch: symmetric encryption (AES), asymmetric encryption (RSA), and advanced hash functions (Perfect Hash Functions). The goal is to discover how these algorithms behave in practice, what implementation challenges arise, and what performance characteristics can be expected.

While theoretical understanding is valuable, implementing these algorithms reveals practical considerations often overlooked: memory alignment, cache efficiency, constant-time operations for security, and trade-offs between code clarity and performance.

0.2.1 Scope

This research implements and analyzes:

- AES-128 with ECB, CBC, and CTR modes of operation
- RSA with 512-4096 bit keys, including CRT optimization
- Perfect Hash Functions using CHD and BDZ algorithms with compression

All implementations are written in C/C++ with comprehensive test suites validating correctness and measuring performance.

0.3 AES-128 Implementation

The Advanced Encryption Standard operates on 128-bit blocks using a 128-bit key with 10 transformation rounds. Each round applies four operations: SubBytes (byte substitution), ShiftRows (row permutation), MixColumns (column mixing), and AddRoundKey (XOR with round key).

Critical Implementation Detail: The state is stored in column-major order in memory. The state array contains 16 bytes indexed 0-15, where indices 0-3 form the first column, 4-7 the second column, and so on. This affects how all transformations are implemented.

0.3.1 Modes of Operation

Three modes were implemented:

- **ECB (Electronic Codebook):** Each block encrypted independently. Not recommended for production.
- **CBC (Cipher Block Chaining):** Each plaintext block XORed with previous ciphertext. Requires IV and PKCS#7 padding.
- **CTR (Counter):** Stream cipher mode encrypting incrementing counters. No padding required.

0.3.2 Performance Results

The implementation passed all 58 test cases including NIST test vectors. Performance metrics:

Mode/Operation	Data Size	Mean Time	Throughput
ECB Single Block	16 bytes	0.0024 ms	511,661 Mbps
CBC Encryption	100 bytes	0.0174 ms	43,746 Mbps
CBC Encryption	1 KB	0.144 ms	54,243 Mbps
CBC Encryption	10 KB	1.257 ms	31,073 Mbps
CTR Encryption	1 KB	0.123 ms	63,490 Mbps
CBC Round Trip	1 KB	0.318 ms	24,557 Mbps

Tabella 1: AES-128 Performance Metrics

Key Observations: CTR mode outperforms CBC due to avoiding padding operations. Throughput decreases with larger data sizes due to cache effects. Key expansion completes in $\approx$**0.36 microseconds**, making it negligible for applications that reuse contexts.

0.3.3 Security Properties

The test suite validates critical security properties:

- **Avalanche Effect:** Single bit changes cause $\approx$50% of ciphertext bits to flip
- **Key Avalanche:** Single bit key changes produce completely different ciphertext
- **IV Uniqueness:** Different IVs produce different ciphertext for identical plaintext

0.4 RSA Implementation

The Rivest-Shamir-Adleman cryptography algorithm is an asymmetric algorithm based on factoring difficulty. This implementation supports 512-4096 bit keys with both standard and CRT-optimized decryption.

Key Generation:

1. Generate large primes p and q using Miller-Rabin testing (40 rounds)
2. Compute modulus $n = p \times q$
3. Compute Euler's totient $\phi(n) = (p-1)(q-1)$
4. Use public exponent $e = 65537$
5. Compute private exponent $d = e^{-1} \bmod \phi(n)$
6. Compute CRT parameters: $d_p = d \bmod (p-1)$, $d_q = d \bmod (q-1)$, $q_{inv} = q^{-1} \bmod p$

0.4.1 Big Number Arithmetic

RSA requires arithmetic on numbers far larger than native CPU types. My custom big number library represents numbers as arrays of 32-bit words, implementing addition, subtraction, multiplication, division, modular reduction, and modular exponentiation.

The core operation is modular exponentiation: $c = m^e \bmod n$ (encryption) or $m = c^d \bmod n$ (decryption), implemented using square-and-multiply with Barrett reduction.

0.4.2 CRT Optimization

The Chinese Remainder Theorem provides significant speedup by computing:

$$m_1 = c^{d_p} \bmod p \tag{1}$$

$$m_2 = c^{d_q} \bmod q \tag{2}$$

$$h = q_{inv}(m_1 - m_2) \bmod p \tag{3}$$

$$m = m_2 + hq \tag{4}$$

This halves the exponentiation bit length, resulting in $\approx$**4$\times$ speedup** theoretically.

0.4.3 Performance Results

All 13 tests passed. RSA-512 performance metrics:

Operation	**Iterations**	**Mean Time**	**Throughput**
Key Generation	10	552.3 ms	1.8 keys/sec
Encryption	50	1.066 ms	22,506 bytes/sec
Decrypt (STANDARD)	50	74.516 ms	322 bytes/sec
Decrypt (CRT)	50	26.528 ms	905 bytes/sec
Signing	50	26.869 ms	558 bytes/sec
Verification	100	1.858 ms	8,074 bytes/sec

Tabella 2: RSA-512 Performance Metrics

Key Findings: CRT achieves **2.81$\times$ speedup** over standard decryption. Key generation varies widely (186-1198ms) due to probabilistic prime finding. Encryption is $\approx$**70$\times$ faster** than decryption because $e = 65537$ has only 2 bits set, while d is a full 512-bit number.

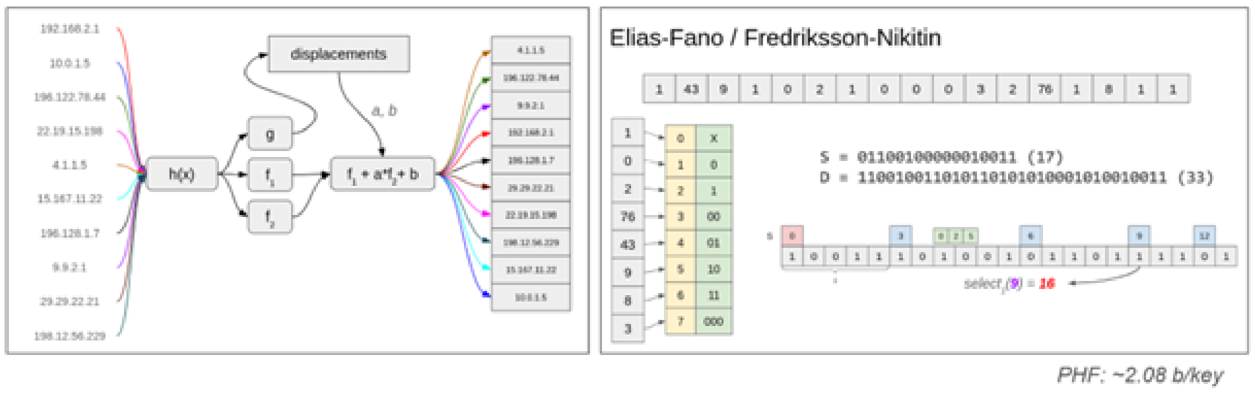
0.5 Perfect Hash Functions

Perfect Hash Functions (PHF) provide collision-free hashing for static key sets. When table size equals key count ($m = n$), the function is a Minimal Perfect Hash Function (MPHF). I implemented CHD (Compress, Hash, and Displace) and BDZ (r-partite hypergraph-based) algorithms.

0.5.1 CHD Algorithm

CHD constructs a PHF by mapping keys into buckets, then displacing collisions using bucket-specific seeds. The algorithm requires $m = (1 + \epsilon)n$ where $\epsilon > 0$ is space overhead ($\epsilon = 0.3$ in the implementation).

CHD - PHF



Compress seeds or displacements by Elias-Fano / Fredriksson-Nikitin coding.
Data table size $(1+\epsilon)n$ for $\epsilon > 0$.
What about k-PHF?

Figura 1: CHD, PHF (source: Algo Talks 3: MPHF, J. Viktorin 2025)

Construction:

1. Hash all keys with $g(x)$ to assign buckets
2. Sort buckets by size (descending)
3. For each bucket, find displacement (a, b) such that $\phi(x) = f_1(x) + a \cdot f_2(x) + b$ maps all keys to unused positions
4. Compress displacement array using Elias-Fano encoding

0.5.2 BDZ Algorithm

BDZ represents the key set as a random r-partite r-uniform hypergraph and checks acyclicity. For $r = 2$ (bipartite), two hash values per key; for $r = 3$, three values.

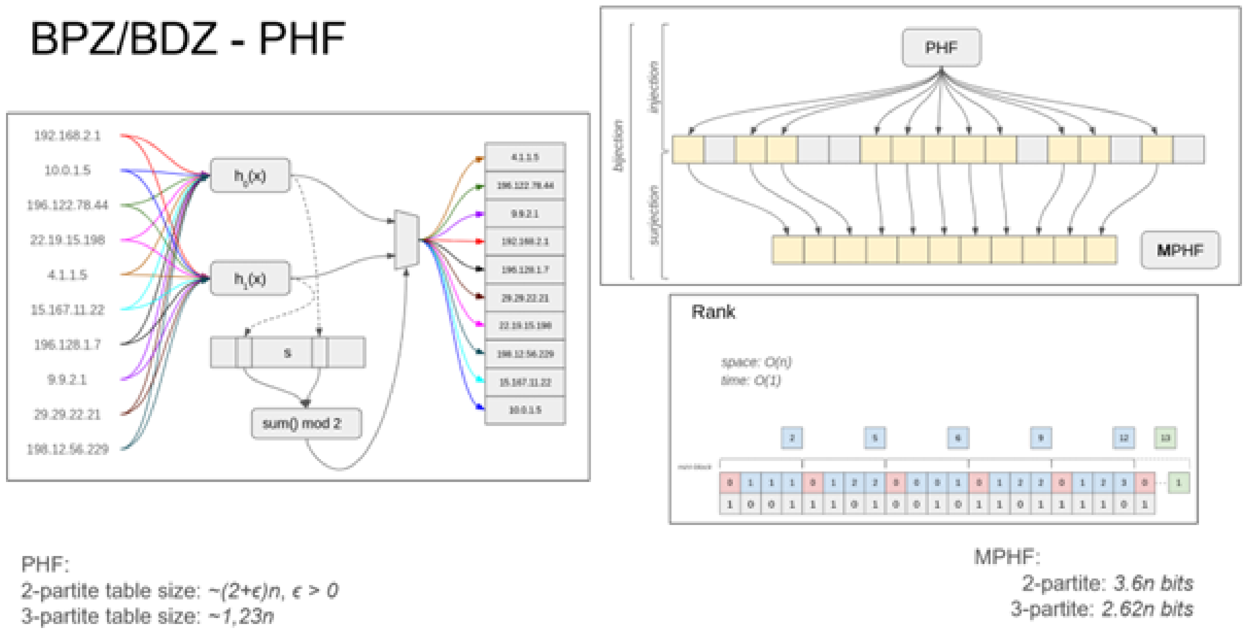


Figura 2: BDZ, PHF (source: Algo Talks 3: MPHF, J. Viktorin 2025)

Construction:

1. Generate r hash values per key, creating hypergraph edges
2. Test acyclicity using vertex degree analysis (O(n) time)
3. If cyclic, regenerate with new seeds
4. Once acyclic, assign values to vertices in reverse topological order
5. Store vertex assignment (g-array) compactly

For BDZ-3, space requirement is $\approx 1.23n$ vertices. The g-array stores values $0..r$, requiring $\lceil \log_2(r+1) \rceil$ bits per vertex.

0.5.3 Compression Techniques

Rank/Select: Given bit vector B, $\text{rank}_1(i)$ counts 1-bits up to position i, $\text{select}_1(k)$ finds the k-th 1-bit. My implementation achieves O(1) rank using two-level prefix sums with O(n) space overhead (succinct).

Elias-Fano: Compresses sorted integer arrays by separating high and low bits. For n integers with maximum M, uses $\approx 2n + n\lceil\log_2(M/n)\rceil$ bits, approaching information-theoretic lower bound. Achieves 3:1 compression on test data (4000 $\rightarrow$ 1376 bytes).

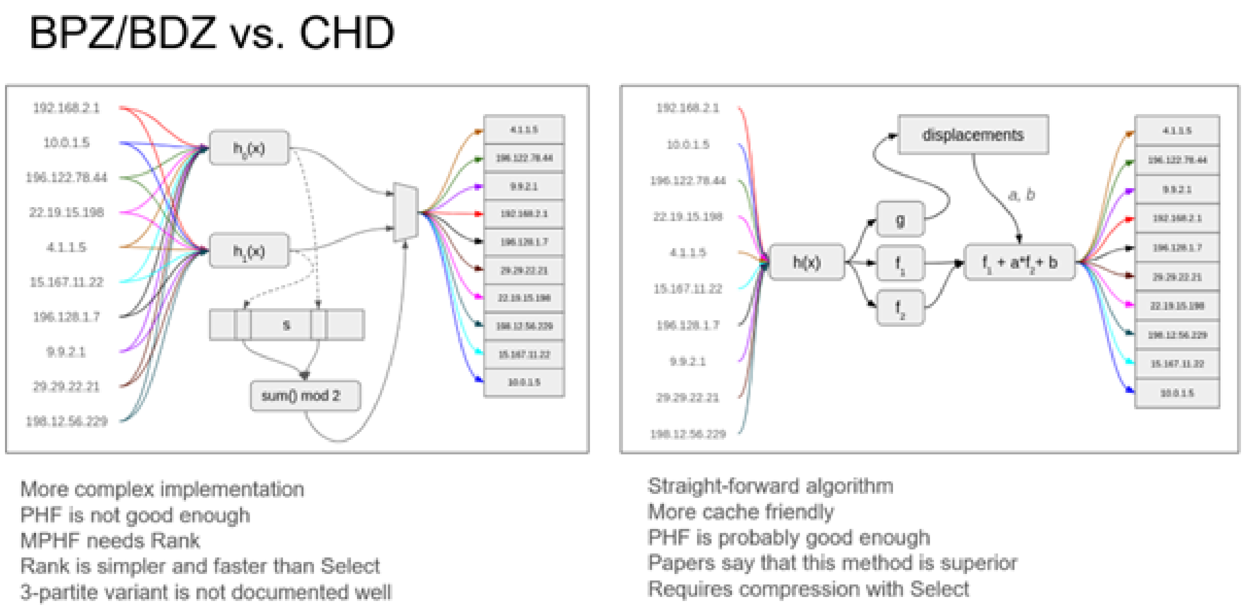
0.5.4 Performance Results

All 20 tests passed. Comprehensive benchmark:

Algorithm	**Keys**	**Build (ms)**	**Query (ns)**	**Memory (KB)**	**Bits/Key**
CHD	100	0.04	79.4	0.38	31.36
BDZ-2	100	0.05	71.7	0.29	23.36
BDZ-3	100	0.08	103.0	0.22	17.76
CHD	1,000	0.51	74.5	3.20	26.18
BDZ-2	1,000	0.92	39.7	2.22	18.18
BDZ-3	1,000	0.88	60.8	1.54	12.58
CHD	10,000	3.30	41.7	31.32	25.66
BDZ-2	10,000	7.06	40.9	21.55	17.66
BDZ-3	10,000	7.08	59.5	14.72	12.06

Tabella 3: Perfect Hash Function Performance Comparison

Key Findings: BDZ-3 achieves best space efficiency **(12-18 bits/key)** vs. **CHD (26-31 bits/key)**. On the other hand, BDZ construction slower due to probabilistic graph generation. All algorithms maintain sub-100ns lookups. Rank/select achieves $\approx$**35ns per query**.



BPZ/BDZ vs. CHD

More complex implementation
PHF is not good enough
MPHF needs Rank
Rank is simpler and faster than Select
3-partite variant is not documented well

Straight-forward algorithm
More cache friendly
PHF is probably good enough
Papers say that this method is superior
Requires compression with Select

Figura 3: Comparison of BDZ and CHD (source: Algo Talks 3: MPHF, J. Viktorin 2025)

0.6 Comparative Analysis

The three implemented cryptographic primitives serve different purposes and operate at vastly different performance scales.

0.6.1 Performance Spectrum

AES operates at throughputs of tens of megabytes per second, making it suitable for encrypting bulk data. RSA is several orders of magnitude slower, processing only hundreds of bytes per second, which is why it's typically used only to encrypt small amounts of data like symmetric keys. Perfect hash functions operate at entirely different timescales queries complete in tens of nanoseconds, but they require precomputation and only work for static key sets.

0.6.2 Practical Applications

These primitives are often combined. Typical secure protocols use RSA to exchange an AES session key (hybrid encryption), leveraging RSA's public-key properties and AES's speed. PHFs, while not cryptographic, enable fast lookup of security policies, IP addresses in firewalls, or domain filtering where key sets are known in advance.

Property	**AES-128**	**RSA-512**	**PHF (BDZ-3)**
Type	Symmetric	Asymmetric	Hash Function
Speed	∼30-60 MB/s	∼300-900 B/s	∼60 ns/query
Key Management	Shared secret	Public/Private	Static key set
Primary Use	Bulk encryption	Key exchange	Fast lookups
Security Basis	Confusion/diffusion	Factoring	N/A (not crypto)
Data Limit	Unlimited	∼53 bytes	N/A
Memory	176 bytes	∼200 bytes	12-31 bits/key

Tabella 4: Comparative Characteristics

0.7 Conclusion

This experimental study successfully implemented three fundamental cryptographic primitives, revealing practical insights:

- **AES-128:** Achieves **30-60 MB/s** in unoptimized C, with CTR outperforming CBC
- **RSA CRT:** Delivers measurable **2.81× speedup**, demonstrating value of mathematical optimizations
- **PHF:** Achieves **12 bits/key (BDZ-3)** with constant-time lookups for static datasets
- **Succinct structures:** Enable practical compression with **$\mathbf{O}(n)$ space** and **$\mathbf{O}(1)$ queries**
- **Performance gap:** AES is ≈**100,000**× faster than RSA, explaining hybrid cryptosystem necessity

Implementing from scratch provided valuable understanding of internal mechanics. Comprehensive testing (NIST vectors, correctness verification, acyclicity testing) was essential. Performance testing revealed constant factors and cache effects matter significantly beyond asymptotic complexity.

This implementation study achieved its goal of providing deep, practical understanding of fundamental cryptographic algorithms. Building these systems from scratch, debugging implementation issues, and analyzing performance provided valuable insights into cryptography.

Bibliografia

[1] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)," FIPS Publication 197, 2001.

[2] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, 1978.

[3] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger, "Hash, displace, and compress," *Proceedings of ESA*, 2009.

[4] F. C. Botelho, R. Pagh, and N. Ziviani, "Simple and space-efficient minimal perfect hash functions," *Proceedings of WADS*, 2007.

[5] P. Elias, "Efficient storage and retrieval by content and address of static files," *Journal of the ACM*, 1974.

[6] J.-J. Quisquater and C. Couvreur, "Fast decipherment algorithm for RSA public-key cryptosystem," *Electronics Letters*, 1982.

[7] J. Viktorin, "Algo Talks 1: Overview over Hash Tables," Presentation slides, 2025.

[8] J. Viktorin, "Algo Talks 2: Perfect Hashing - CHD & BPZ/BDZ," Presentation slides, 2025.

[9] J. Viktorin, "Algo Talks 3: MPHF," Presentation slides, 2025.