



Paradox: Recursive Visual Entropy Key Derivation Engine (RVE-KDE) Experimental Framework for Deterministic Image Based Cryptographic Key Generation

Chirag Ferwani¹, Suvarna Bhatsangave², Sarika Bartakke³, Vrushabh Hirap⁴,
Kshitij Jadhav⁵, Anushka Shinde⁶

^{1,2,3,4,5,6} Department of Computer Science and Engineering (AIML), Pimpri Chinchwad College of Engineering, Pune, Maharashtra, India.

To Cite this Article: Chirag Ferwani¹, Suvarna Bhatsangave², Sarika Bartakke³, Vrushabh Hirap⁴, Kshitij Jadhav⁵, Anushka Shinde⁶, "Paradox: Recursive Visual Entropy Key Derivation Engine (RVE-KDE) Experimental Framework for Deterministic Image Based Cryptographic Key Generation", Indian Journal of Computer Science and Technology, Volume 05, Issue 02 (May-August 2026), PP: 768-804.



Copyright: ©2026 This is an open access journal, and articles are distributed under the terms of the [Creative Commons Attribution License](#); Which Permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Abstract: In the past, cryptographic key derivation has been achieved by using a password, a random number generator, or a hardware-based entropy source to generate secure encryption keys. The authors introduce a novel framework, called Paradox: Recursive Visual Entropy Key Derivation Engine (RVE-KDE), which presents an experimental solution to the problem of exploring digital images as an alternative deterministic source of cryptographic entropy. The proposed method is based on recursive image traversal, pixel-level visual entropy extraction and state evolution using hash-chains to generate reproducible cryptographic keys from image data. Unlike conventional processing methods, which process linear inputs, Paradox recursively examines multiple areas of a picture to gather color, spatial and structural data and incorporates it into an entropy pool. This entropy is then passed through a key derivation step to create deterministic symmetric keys that can be used with existing and proven encryption methods like AES-256-GCM and ChaCha20-Poly1305. To test the proposed framework, a Python implementation was created and experimentally evaluated using avalanche effect analysis, Hamming distance distribution, collision testing, latency measurements, and a comparison against several other key derivation functions (KDFs) such as PBKDF2, HKDF, Argon2id, and BLAKE3. Experimental results show deterministic repeatability and statistical properties matching the assessed cryptographic parameters. Paradox is not designed to be the replacement for standard key derivation function, but rather a framework for research in visual entropy as a new input space and a reproducible base for further studies in image based cryptographic key derivation.

Keywords: Cryptography, Key Derivation Functions (KDF), Visual Entropy, Recursive Image Traversal, Hash Chain Evolution, Deterministic Cryptographic Key Generation, Image Processing, Symmetric Encryption.

I. INTRODUCTION

Every time a file is encrypted, a message is secured, or a financial transaction is authenticated, the process ultimately rests on a single foundational element: the cryptographic key. The strength of any encryption system is only as good as the key it uses. If the key is weak, predictable, or derived from a low-entropy source, the entire security chain is compromised regardless of how sophisticated the underlying algorithm is. This reality has driven decades of research into key derivation functions - algorithms specifically designed to transform input material, typically a password or a random seed, into a cryptographically strong symmetric key. Today, the most widely deployed key derivation functions – PBKDF2 [1], HKDF [2], scrypt [3], Argon2id [4], and BLAKE3-KDF - are mature, well-studied, and standardized. [1] Each of them approaches the problem differently: PBKDF2 applies iterative hashing to slow down brute-force attacks, scrypt and Argon2id introduce memory-hardness to defeat GPU-based cracking, and HKDF provides a clean extract-and-expand paradigm for deriving keys from already-strong material. Despite their differences, these systems share a common dependency: they all ultimately rely on the entropy quality of their primary input, and in practice, that input is often a human-generated password.

This is where a fundamental tension emerges. Human beings are poor sources of cryptographic entropy [5]. Passwords are short, reused across services, drawn from predictable patterns, and heavily influenced by memory limitations. Studies consistently show that even when users attempt to create complex passwords, the actual entropy of the resulting strings falls far short of what cryptographic security demands [1], [5]. Salting, stretching, and memory-hardening can compensate to a significant degree, but they cannot manufacture entropy that was never present in the original input. This observation raises a natural question that appears surprisingly underexplored in the cryptographic literature: if passwords represent a low-dimensional, low-entropy input, what happens when we substitute a fundamentally richer source - such as a digital image.

A modern photograph is not merely a collection of pixels. It is a high-dimensional information structure containing millions of individual color measurements, local spatial relationships, luminance gradients, textural variations, edge patterns, and geometric arrangements. Two photographs that appear visually similar to a human observer can differ in tens of thousands of pixel values at the byte level. A minor change in lighting, camera angle, or scene content produces a cascade of differences distributed across the

entire image matrix. In raw information-theoretic terms, a high-resolution image carries orders of magnitude more entropy than a typical human-chosen password, and it does so in a form that is both structured and unique.

What remains largely unstudied is whether this visual richness can be transformed into deterministic cryptographic key material not by treating the image as a hash input, which would collapse all spatial structure into a single fixed-size digest, but instead by traversing the image recursively, building up entropy through an evolving path that depends on the image's own content at each step.

This paper introduces Paradox: Recursive Visual Entropy Key Derivation Engine (RVE-KDE) [6], an experimental framework designed to investigate exactly this question. Rather than hashing an image in a single pass, Paradox treats the image as a traversal space. It begins from a deterministically computed starting coordinate, walks across the image by following a path shaped by a continuously evolving cryptographic hash chain, and collects entropy from each visited pixel and its immediate spatial neighborhood. This process is repeated across multiple dependent recursive layers, each one beginning where the previous layer ended, so that the final entropy pool reflects a deep, multi-pass exploration of the entire image space. The resulting pool is then processed through HKDF-SHA256 [2] or BLAKE3-KDF to produce a symmetric key of the desired length - 128, 256, or 512 bits that is fully compatible with AES-256-GCM [7] and ChaCha20-Poly1305 [8] authenticated encryption.

The same image, combined with the same nonce and security configuration, always produces the same key. A different image, a different nonce, or even a single modified pixel produces a completely different key. This determinism is what makes the system usable in practice: a sender and receiver who share a reference image can independently reconstruct the same key without any direct key exchange, because the key derivation process is fully reproducible given identical inputs.

It needs to be clearly stated upfront that Paradox is not proposed as a production replacement for Argon2id [4], HKDF [2], or any standardized key derivation function. The system has not been formally proven secure, has not undergone adversarial cryptanalysis by external parties, and carries known limitations - most critically, the absence of nonce-misuse resilience. The contribution of this work is explicitly experimental: to design, implement, document, and rigorously evaluate a novel image-driven key derivation mechanism, and to honestly report what it does well, where it falls short, and what questions it opens for future research.

I.I. Motivation

The motivation for Paradox comes from a fairly simple observation: the field of key derivation has spent considerable energy on the question of how to derive keys, but comparatively little on the question of what to derive them from. The standard answer - passwords, shared secrets, hardware randomness reflects decades of practical deployment, not necessarily the full space of what is possible. Digital images represent one of the most abundant forms of information produced by modern computing systems. Billions of images are captured, stored, and transmitted every day across consumer devices, scientific instruments, satellite systems, medical equipment, and social platforms. Each one of these images contains a unique, high-dimensional structure that has, to date, been studied primarily for its visual content rather than its informational potential as an entropy carrier. [2]

The idea behind Paradox is to reframe the image not as data to be encrypted, but as a participant in the encryption process itself a source of traversable entropy whose specific structural characteristics drive the path and evolution of the key derivation computation [1], [2]. If this reframing can be made deterministic and reproducible, it opens the door to a new class of key derivation mechanisms in which the "key material" is effectively encoded in the visual structure of a shared reference image. [3]

Beyond the cryptographic framing, there is also an educational and exploratory motivation. Paradox was designed to be an open, transparent, and rigorously tested research prototype - one that allows the broader community to inspect its design decisions, reproduce its results, examine its weaknesses, and build on its ideas. [4], [9] The codebase is publicly available, the benchmark suite is fully documented, and the experimental results are reported without omitting the areas where the system underperforms relative to conventional alternatives.

I.II. Problem Statement

Current key derivation functions handle the transformation of entropy into key material with remarkable effectiveness. Their designs are well-understood, their security properties are formally analyzed, and their implementations are battle-tested across billions of real-world deployments. The problem this paper addresses is not a failure in these existing systems - it is the absence of a deterministic framework capable of exploiting the high-dimensional entropy present in visual data. [1], [5] Specifically, three distinct gaps motivate this work. First, no widely adopted key derivation framework exists that uses recursive image traversal as its primary entropy mechanism. Second, the question of whether multi-layer recursive walks across image space produce entropy pools with statistically sound cryptographic properties has not been systematically studied. [7] Third, no publicly available implementation exists that allows researchers to directly compare image-driven recursive key derivation against standard KDFs on identical benchmarks in a reproducible and open experimental setting.

Paradox is presented as a first attempt to address all three of these gaps simultaneously: through system design, implementation, and empirical evaluation.

I.III. Research Objectives

The research presented in this paper pursues three clearly defined objectives.

- The first objective is to design and implement a functional, deterministic image-based key derivation framework that uses recursive visual traversal as its primary mechanism for entropy extraction. The design must be fully reproducible given identical inputs, the system must always produce identical outputs across platforms and executions. [10]

- The second objective is to investigate whether recursive hash-chain evolution and multi-layer image traversal produce entropy pools with statistically valid cryptographic properties. This includes measuring Shannon entropy [5], byte uniformity through chi-square analysis [11], avalanche sensitivity [12], Hamming distance distributions [13], and collision rates across large key generation samples. [8]
- The third objective is to benchmark the proposed framework against PBKDF2-SHA256, HKDF-SHA256, Argon2id, and BLAKE3-KDF [9] on key generation latency, memory consumption, throughput, image-sensitivity, and overall cryptographic quality and to report the results honestly, including both the areas where Paradox performs comparably and the areas where it does not.

I.IV. Key Contributions

This work makes the following concrete contributions to the research literature:

The primary contribution is the design and open-source implementation of the Recursive Visual Entropy Key Derivation Engine (RVE-KDE) [6] a novel framework that derives cryptographic keys from images through deterministic recursive traversal, hash-chain state evolution [14], and multi-layer entropy aggregation [15].

A secondary contribution is the development of a recursive image traversal mechanism that replaces sequential pixel scanning with a coordinate-guided walk driven by the hash chain's own evolving state ensuring that the traversal path is both unpredictable to an observer and reproducible for a legitimate user. [8] An additional contribution is the systematic experimental validation of the framework across multiple security configurations, image types, and key lengths, using a reproducible benchmark suite that includes all standard cryptographic quality metrics. [5], [11], [12], [13]

Finally, the work provides an honest and comparative assessment of the framework's position relative to established key derivation functions, identifying precisely where visual entropy adds distinctive research value and where the current design requires further development. [4]

I.V. Scope and Limitations

Paradox is an experimental research system. Everything about its design, evaluation, and presentation in this paper is oriented toward the goals of academic investigation not production deployment. This distinction matters and needs to be stated clearly from the outset.

The framework has not been submitted to independent security auditing, has not been formally proven secure under standard cryptographic models, and does not implement the memory-hardness mechanisms that protect systems like Argon2id against GPU-accelerated brute-force attacks. [1] The nonce-misuse vulnerability described in detail in Section 9 is a serious structural limitation: reusing the same image and nonce pair will always produce the same key, with no built-in countermeasure. These are known, documented, and honestly reported limitations [16] and not oversights.

The security analyses presented throughout this paper are limited to statistical and empirical evaluation: entropy measurement, chi-square uniformity testing, avalanche effect analysis, and comparative latency and memory benchmarking. Formal cryptographic proofs, adversarial cryptanalysis, and game-based security reductions remain outside the scope of the current work and are explicitly identified as necessary future research directions before any production use of the framework could ever be considered. Within its intended scope studying recursive visual entropy as a novel key derivation mechanism in an open, reproducible, and honestly evaluated research setting, Paradox represents a meaningful first contribution to a largely unexplored design space.

II.BACKGROUD

A) Key Derivation Functions

Before understanding what Paradox does differently, it helps to understand what key derivation functions are, why they exist, and what problems each of the major ones was built to solve. This context matters because Paradox does not operate in isolation it sits on top of these systems, and understanding their design philosophy makes it easier to appreciate both what the framework borrows from them and where it diverges. At its core, a key derivation function takes some input a password, a shared secret, a randomly generated seed - and transforms it into a cryptographic key of a specific length and format that can be used for encryption, authentication, or signing. This sounds simple, but the engineering challenge underneath it is significant.

The input to a KDF is rarely perfect. Human-chosen passwords are short, reused, and drawn from predictable patterns. Shared secrets generated during a network handshake may be mathematically sound but may not be in the exact format required by a symmetric cipher. Raw entropy from a hardware source may need to be stretched into multiple independent keys for different purposes. In each of these scenarios, the KDF is doing something more nuanced than just hashing its input - it is conditioning that input into something cryptographically usable, regardless of how imperfect or low-entropy the original material was.

The defining properties that every well-designed KDF must satisfy are determinism and sensitivity. Given identical inputs, the function must always produce the identical output otherwise decryption would be impossible. At the same time, even the smallest change to the input must produce a completely different output this is the avalanche property, and without it the function leaks information about its input. These two requirements exist in natural tension and sit at the heart of how every KDF is designed.

PBKDF2 [1], [17] - Slowing the Attacker Down. PBKDF2 was one of the first widely standardized approaches to the problem, and its design logic is almost elegantly simple: if an attacker is trying to guess a password by testing millions of candidates, make each test as expensive as possible. The way PBKDF2 achieves this is through deliberate repetition. It takes the

password and salt, runs them through a pseudorandom function - typically HMAC-SHA256 or HMAC-SHA512 - and then runs that output through the same function again, thousands or even hundreds of thousands of times. The final output after all those iterations becomes the derived key. The legitimate user, [7], [8] who only needs to derive the key once, barely notices the overhead. The attacker, who needs to repeat this process for every single password guess, faces a computation cost that scales directly with the iteration count.

This was a genuinely effective defense when it was introduced. The problem is that computational hardware has not stood still. Modern GPUs can execute SHA-based operations at extraordinary speed, and specialized password-cracking hardware like FPGAs and ASICs can run even faster. Increasing the iteration count helps, but it is ultimately a race against hardware throughput - and it is a race that pure computational delay tends to lose over time. PBKDF2 remains widely deployed and is not broken, but the cryptographic community recognized early that a more robust defense would need to involve something hardware cannot simply accelerate: memory.

HKDF [2] - When the Input is Already Strong HKDF was designed to solve a different problem entirely. While PBKDF2 is concerned with strengthening weak passwords, HKDF assumes its input is already cryptographically strong - the kind of material that comes out of a Diffie-Hellman key exchange, for instance, or a previous KDF output. The challenge HKDF addresses is not strengthening weak material but securely expanding strong material into one or more independent keys of arbitrary length.

It operates in two stages. The first stage, called Extract, takes the input key material and a salt and condenses everything into a fixed-length pseudorandom key using HMAC. This step normalizes the input, ensuring it has the right format and entropy distribution regardless of where it came from. The second stage, called Expand, takes that pseudorandom key and derives as many independent output keys as needed, each tagged with a context label that separates them from one another.

This two-stage design is what makes HKDF so well suited to modern communication protocols. TLS, QUIC, the Signal protocol, and many other systems need to derive multiple independent session keys from a single shared secret - one key for encryption in each direction, one for authentication, one for integrity checking, and so on. [2] HKDF handles all of that cleanly and provably, with formal security analysis backing its design. The tradeoff is that it is genuinely not designed for passwords. Feed a weak, predictable password directly into HKDF and it will derive a key efficiently, but the key will inherit all the weakness of the original input. HKDF does not compensate for entropy that was never there.

scrypt - Making Memory the Bottleneck. The insight behind scrypt is that while computational speed is something attackers can buy more of at a reasonable price, memory bandwidth is fundamentally harder to scale in parallel. [3] A GPU with thousands of cores can run thousands of SHA computations simultaneously. But if each computation requires gigabytes of memory that must be read and written in a specific access pattern, the attacker cannot simply add more cores they need to add more memory for each core, and memory does not parallelize the same way that computation does. scrypt exploits this asymmetry by requiring its computation to continuously allocate and access large regions of memory throughout execution. The access pattern is deliberately designed to be sequential and dependent on previous steps, meaning the computation cannot be efficiently split across parallel units without each unit maintaining its own full memory allocation. This makes scrypt genuinely expensive to attack with GPUs or FPGAs in a way that PBKDF2 never fully achieved.

The tradeoff is operational. On constrained devices embedded systems, older mobile hardware, low-memory servers the memory requirements of scrypt can become a practical burden for the legitimate user as well. Getting the parameters right for a specific deployment requires careful benchmarking, and a misconfigured scrypt can either provide insufficient protection or make the authentication process noticeably slow. Despite this, scrypt remains a respected and widely deployed algorithm, and its memory-hard design philosophy directly influenced everything that came after it.

Argon2id - The State of the Art. Argon2 won the Password Hashing Competition in 2015 after years of open evaluation by the global cryptographic community, and its recommended variant Argon2id represents the clearest current answer to the question of what a password-based KDF should look like. [18] What makes Argon2id distinctive is how deliberately it was designed for tunability. Three parameters control its behavior: memory, time, and parallelism. The memory parameter controls how much RAM the computation occupies. The time parameter controls how many passes it makes over that memory. The parallelism parameter controls how many independent threads can cooperate in the computation. Together, these three knobs allow a developer to calibrate the algorithm precisely for their deployment environment - a high-security server can be configured to use gigabytes of RAM and multiple seconds of computation time, while a mobile application can use a smaller memory footprint with less delay, all using the same algorithm with different parameters.

Argon2id [4] also combines two access patterns from its sibling variants. Argon2i uses data-independent memory access, which resists side-channel timing attacks but is somewhat weaker against time-memory tradeoffs. Argon2d uses data-dependent access, which is stronger against tradeoffs but potentially leakable through timing. Argon2id begins with data-independent access and switches to data-dependent access partway through, capturing the benefits of both. The result is a function that is memory-hard, timing-attack resistant, formally analyzed, and recommended by virtually every major security organization for new password-storage deployments. For this paper, it represents the strongest conventional baseline against which Paradox is benchmarked.

BLAKE3 - When Speed is the Priority. BLAKE3 occupies a different position in this landscape. It is not primarily a password-hardening function. It does not intentionally slow itself down or consume large amounts of memory. Its entire design philosophy points in the opposite direction: BLAKE3 is built to be as fast as possible while maintaining strong cryptographic guarantees.

The key architectural decision behind BLAKE3 is the use of a Merkle tree structure. Instead of processing input as a single sequential chain, BLAKE3 splits the input into chunks, hashes each chunk independently, and then combines the results in a binary tree. This means that on a multi-core processor, all those chunk-level computations can run in parallel simultaneously. On modern

hardware with SIMD instruction sets, BLAKE3 processes data at speeds that would have seemed implausible for a cryptographic hash function a decade ago.

BLAKE3 [9] provides a key derivation mode as part of its specification, which allows it to be used as a fast, high-throughput KDF for applications where the input is already trusted. This is where Paradox uses it as a final compression step that converts the accumulated entropy pool into a well-formatted cryptographic key. It is not being used here as a password-hardening mechanism. It is being used because it provides an extremely efficient, well-analyzed pathway from an entropy pool to a properly formatted symmetric key.

Looking across all five of these algorithms, a clear pattern emerges. Each one was designed to solve a specific problem. PBKDF2 slows attackers down through repetition. HKDF expands strong material cleanly and provably. scrypt makes memory the bottleneck instead of compute. Argon2id combines memory hardness with formal analysis [19] and tunable deployment. BLAKE3 prioritizes performance and parallelism without sacrificing cryptographic soundness.

Despite their architectural diversity, every one of these systems treats its entropy input the same way: as a one-dimensional string of bytes, whether that string comes from a password, a handshake result, or a random generator. None of them interact with the structural or spatial properties of their input in any meaningful way. The notion of an image -with its millions of pixels, local spatial relationships, color distributions, and geometric patterns - simply does not appear anywhere in their design space. [20]

This is not a criticism of these systems. They were designed for their specific purposes and they do those things well. But it does identify a genuine gap in the literature, and it is precisely that gap that Paradox was built to investigate. Not to replace any of these functions, but to ask a question they were never designed to answer: what happens when the entropy source is not a string, but a traversable two-dimensional space.

B) Entropy Sources in Cryptography

Before examining how Paradox extracts entropy from images, it is worth stepping back and understanding what entropy actually means in a cryptographic context and why the quality of an entropy source matters so fundamentally to the security of everything built on top of it. In cryptography, entropy refers to the degree of unpredictability present in a source of information. A high-entropy source produces outputs that an attacker cannot reasonably predict or reproduce. A low-entropy source, regardless of how sophisticated the algorithm processing it might be, produces keys that are weaker than they appear because the real search space an attacker needs to explore is not the theoretical key space, but the much smaller space of likely inputs that could have produced those keys. Every bit of a cryptographic key needs to be genuinely unpredictable, and achieving that consistently across real-world systems turns out to be harder than it sounds. [21]

Random Number Generators sit at the foundation of nearly every cryptographic protocol in existence. These systems supply the raw unpredictable values that become keys, salts, nonces, initialization vectors, and other security-critical parameters. They generally fall into two categories. True Random Number Generators, [22] or TRNGs, harvest randomness from physical phenomena thermal noise in electronic components, atmospheric interference, radioactive decay, oscillator jitter, the unpredictable timing of hardware interrupts. Because these sources are grounded in physical reality rather than mathematical computation, they provide exceptionally high-quality entropy that is genuinely impossible to predict from first principles. The tradeoff is that they are slow, require specialized hardware, and produce randomness at a rate limited by the underlying physical process rather than the speed of a processor. Pseudorandom Number Generators, or PRNGs, take a different approach entirely they use a deterministic mathematical algorithm seeded with a small amount of initial randomness and expand that seed into a much longer sequence of values that pass statistical tests for randomness. A well-designed cryptographic PRNG is computationally indistinguishable from a true random source by any efficient algorithm, which makes them practical for nearly every use case. The vulnerability is in the seed: if an attacker can discover or predict the initial seed, every output the generator ever produces becomes reconstructible from that point forward. [23]

Hardware entropy sources represent the closest thing to a solved problem that the field of randomness generation has produced. Modern processors increasingly include dedicated on-chip hardware modules that continuously harvest entropy from physical processes occurring within the silicon itself, entirely independent of software. Intel's RDRAND and RDSEED

[20] instructions expose this directly to application code. Trusted Platform Modules provide a hardware-rooted entropy source that is architecturally isolated from the main processor. Hardware Security Modules used in enterprise environments go further, providing physically tamper-resistant entropy generation certified under formal security standards. The advantage of these systems is that they do not depend on user behavior, application state, or anything an attacker can influence through software. Their randomness originates at the physical level, which puts it outside the threat model that most software-based attacks operate within. The limitations are equally real: hardware entropy sources add cost, are not universally available across all device categories, and must themselves be carefully validated a manufacturing defect or implementation flaw in a hardware RNG can silently degrade entropy quality in ways that are extremely difficult to detect after deployment. [24] Operating system entropy pools represent the practical middle ground that most applications actually rely on. Rather than exposing hardware entropy directly, modern operating systems Linux, Windows, macOS maintain internal entropy pools that continuously collect randomness from dozens of independent sources simultaneously: keyboard timing intervals, mouse movement deltas, disk access latencies, network packet arrival times, processor execution jitter, hardware interrupts, and dedicated hardware RNG outputs where available. All of this is mixed together into a single entropy pool, and applications request random values through a secure system interface /dev/urandom on Linux, CryptGenRandom on Windows without ever needing to manage individual entropy sources themselves. This architecture is both practical and robust. Even if one or two contributing sources temporarily produce poor-quality randomness, the pool's aggregate entropy remains high because the sources are independent. The operating system handles all of the low-level complexity, and application developers get a clean, high-quality randomness interface they can

use without expertise in hardware entropy collection.

Password entropy [17] is where the picture becomes considerably less optimistic, and understanding why matters directly for motivating the research direction that Paradox explores. Unlike hardware-generated randomness, passwords are created by human beings and human beings are fundamentally poor at generating unpredictability. Passwords tend to follow linguistic patterns, incorporate personally meaningful words or dates, substitute characters in predictable ways, and cluster around common patterns that repeat across millions of users independently. Large-scale password leak analyses have demonstrated repeatedly that real-world password distributions are dramatically more concentrated than their theoretical character-space entropy would suggest. A password drawn from a 94-character alphabet and eight characters long has a theoretical entropy of roughly 52 bits but the actual distribution of passwords humans choose from that space has an effective entropy far lower, because the space is not explored uniformly. Key derivation functions like PBKDF2, scrypt, and Argon2id address this by increasing the cost of testing each candidate, but they cannot create entropy that was never in the input. They make attacks slower; they do not make bad passwords good. [21]

Alternative and unconventional entropy sources have attracted growing research interest as the community has explored ways to supplement or replace password-derived entropy with richer inputs. Behavioral biometrics keystroke timing dynamics, mouse movement patterns, touchscreen gesture characteristics, gait signatures measured through accelerometers have been studied as entropy contributors for authentication and key binding systems. Wireless channel characteristics, which are unique to the physical environment between two communicating devices, have been investigated as a shared entropy source for key agreement without explicit communication. Physiological measurements including electroencephalography and cardiac rhythms have also appeared in the literature as potential sources of user-specific cryptographic material. Most of these approaches face practical challenges around reproducibility, measurement noise, and environmental sensitivity that have prevented their adoption as primary entropy sources in standardized systems. Digital images occupy an interesting position in this landscape. Unlike passwords, an image naturally contains millions of pixels distributed across a complex, high-dimensional spatial arrangement of colors, textures, gradients, and local structural patterns. The sheer information content of even a modestly sized photograph dwarfs that of any practical password. Yet despite this, research involving images in security contexts has concentrated almost entirely on encryption, watermarking, authentication, and visual cryptography not on deterministic key derivation that treats the image's own pixel structure as the entropy source itself.

This observation defines the research gap that Paradox is built around. The following section examines the prior work that comes closest to addressing it visual cryptography, perceptual hashing, image fingerprinting, and image-based steganographic systems and identifies precisely where the existing literature stops short of what a deterministic image-driven key derivation framework would require. [25]

C) Image as an Entropy Source

Images are one of the most richest forms of information in digital form because of the swift development of digital imaging technologies. Modern images are made up of millions of picture elements called pixels, each of which has several channels of information that are organized in a structured space. Images do have high dimensional properties color distributions, texture variations, edge properties, contrast gradients, spatial relationships, and geometric patterns that do not exist in conventional textual inputs like password or binary secret. These properties together create a very complex information space, and the attention to it has been growing in computer vision, image processing and cybersecurity. Digital images have been used for decades in multimedia applications but the use of this medium as a deterministic source of cryptographic entropy is relatively under-explored. [26]

Historically, research in images used in cryptography has been concerned with the protection of the visual information, rather than the use of visual information itself for key generation. Among the most important ideas in this area is visual cryptography, in which a secret image is split into several parts, each of which is an innocuous image but when combined can reconstruct the original image. These techniques mainly focus on secure image distribution, secret sharing and authentication, without generating cryptographic keys from the image content. This means that the image is not the entropy source used for key generation, but rather is the object being protected. [27]

Another related field of research is image fingerprinting, which aims to create “fingerprints” that are small digital signatures that uniquely identify an image. Instead of storing the value of each pixel, fingerprinting algorithms identify a few more salient features that are relatively constant when common transformations like compression, resizing or minor noise occur. With the function of efficiently comparing between large collections of images, image fingerprinting has become more and more important for copyright protection, duplicate image detection, digital forensics and multimedia indexing. They are however techniques for maximizing the accuracy of the recognition rather than maximizing entropy, so they are not suitable as cryptographic key generation mechanisms. [28]

Perceptual hashing is similar, except that the images to be compared are deliberately altered to create a similar hash. [29] Perceptual hashes are not conventional cryptographic hash functions that vary greatly between even minor changes in a single bit of the image, but rather they try to maintain semantic similarity among similar images. Algorithms like average hashing, difference hashing, and perceptual hashing can reduce image content to a small signature, which can be used to quickly locate images with similar properties and to detect image duplicates. While these techniques can effectively provide a summary of visual information, their goal is very different from cryptographic key derivation, since similarity preservation is in direct conflict with the unpredictability necessary to cryptographically secure key generation.

In computer vision, researchers have delved deep into image feature extraction methods, in addition to image hashing. Corner, edge, gradient, texture, color histogram and local descriptors are algorithms that are critical in object recognition, scene

understanding, image retrieval and pattern classification systems. The methods show that digital images possess much structural information spread across several spatial scales. [30] Feature extraction algorithms view images not as just pixels, but as patterns of relationships between neighborhoods and image structure. This multi-dimensional representation is indicative of the fact that the digital image contains a significant informational level of complexity in its natural form.

More recently, image representations have been shown to be expressive using machine learning and deep learning. CNNs learn hierarchical visual features on their own that are able to classify very complex objects and scenes without requiring feature engineering. These representations have shown their great effectiveness at improving image recognition and classification performance, but are still largely focused on semantic understanding as opposed to cryptographic entropy generation. They are looking not for the most random, or unpredictable, image, but rather for a meaningful pattern within the picture. [29]

Some image encryption methods have proposed embedding image features into keys to be used during encryption, such as selecting a subset of image features or using pixel statistics or chaotic mathematical systems. Many of these techniques involve adding more complexity to the encryption process, but some require image-specific transformations which are not deterministically repeatable on independent systems, or rely on external parameters. In addition, many image encryption methods aim at securing the actual image, while they do not provide a general scheme that can be used to generate reusable cryptographic keys for other types of data, like text files, documents or binary data. [31]

Digital images have a number of characteristics that make them desirable targets for exploration of their entropy from an information-theoretic point of view. Each pixel carries several channels of data, usually in the form of the red, green and blue intensities. The neighboring pixels add further spatial dependencies in the form of textures, gradients, edges and local contrast variations. Together, these attributes make the state space very large and rapidly increase as image resolution increases. Relatively small images have hundreds of thousands of independent values that are organized into structures. This multidimensional information not only gives much richer input than traditional textual passwords, but is also completely deterministic for identical images.

Interpretation of the raw pixel values as cryptographic entropy, however, presents a number of important challenges. There is a strong correlation between neighboring pixels in natural images, as their colors and intensities are likely to be similar. [30] One might get a very small number of significant bits of information from a large, homogeneous region, like a sky or a wall or a background, even though it may have millions of pixels. Therefore, it is not possible to extract the entropy by reading the pixel values sequentially alone. Rather, any useful entropy generation depends on the proper design of traversal strategies that can be used to explore the various regions of image space and that are continually modified to remove spatial redundancy.

This is the motivation behind the recursive traversal of images. An image can be scanned recursively, with the exploration path changing dynamically based on information already gathered. The visited sites affect the decisions made for further traversal, allowing richer and richer interaction with the underlying visual structure. Recursive traversal, in conjunction with cryptographic state evolution, offers a way to convert locally correlated image data to increasingly complex entropy sequences without losing determinism of the evolution.[32]

Although individual components of this idea such as image processing, feature extraction, perceptual hashing, and cryptographic hashing have each been studied extensively, existing literature provides little evidence of an integrated framework that combines recursive image traversal, evolving cryptographic state, and deterministic visual entropy extraction for generalized key derivation. [33] Most existing research either treats images as protected data, extracts compact fingerprints for identification purposes, or applies conventional cryptographic algorithms to encrypt image content. Very few approaches investigate images themselves as deterministic entropy domains capable of driving cryptographic key generation.

The proposed Paradox framework is motivated by this observation. Instead of replacing established key derivation functions, it investigates whether recursive analysis of image content can serve as a deterministic preprocessing stage that generates a high-dimensional entropy pool before conventional cryptographic key derivation is performed. [34] By integrating recursive traversal, pixel-level entropy extraction, and hash-chain state evolution into a unified pipeline, Paradox seeks to bridge the gap between image processing and cryptographic key derivation while maintaining compatibility with existing encryption standards. This perspective distinguishes the proposed framework from prior image-based security approaches and establishes the foundation for the architectural design presented in the following sections.

D) Hash Chain Constructions

Hash chains have become the backbone of evolving security, serving as one of the most brilliant and fundamental tools in modern cryptography. Instead of treating every security operation as an isolated event, hash chains create a continuous, evolving storyline where every new cryptographic state is directly born from the one that came before it. Because of this recursive relationship, every new state carries the fundamental makeup of the entire chain's history. [35] This makes the chain highly predictable if you possess the key, but computationally impossible to guess if you do not. As a result, hash chains have become the gold standard for secure messaging, blockchains, and future-proof key management.

At its heart, a hash chain relies on a remarkably simple principle: an initial secret is run through a cryptographic hash function, and that output becomes the input for the next round. Because modern hash functions possess a strong avalanche property, changing even a single microscopic detail in an early state will completely scramble all future outputs. This gives hash chains a unique superpower, making them perfectly reproducible if you know the exact starting point, yet incredibly resistant to being predicted or reverse-engineered by an outsider. [36]

Developers likely rely on hash chains every day without realizing it. For instance, Hash-based Message Authentication Code, or HMAC, [2] uses this evolving internal state to generate multiple independent security keys from a single master password without needing an outside source of randomness. Secure messaging apps use similar ratchet mechanisms, like the Signal Protocol,

to change your encryption key after every single message. As the conversation moves forward, old keys are permanently destroyed and replaced by new ones. This guarantees forward secrecy, meaning that past messages remain locked away even if a current key is stolen. Furthermore, blockchains and secure logging systems use hash chains to tie previous data blocks to new transactions, creating an immutable digital audit trail where altering a past record instantly breaks the entire chain.

The true beauty of an evolving hash chain lies in its ability to accumulate context over time. As the chain moves forward, it does not merely process the next input in isolation; it can fold in timestamps, message contents, and environmental data. Consequently, the cryptographic state gradually becomes a compressed, highly secure history book of everything that has happened so far. [37] This provides a perfect balance, remaining fully deterministic so you can reproduce it exactly when needed, while staying computationally irreversible so bad actors cannot work backward to uncover previous states.

Despite how powerful hash chains are, almost all current systems feed them basic, one-dimensional data like passwords, text messages, or binary codes. While the concept of evolving a cryptographic state is well-established, using the chaotic, rich data of high-dimensional media like digital images to drive that evolution remains largely unexplored territory. [38] This is where the Paradox framework steps in, taking the proven philosophy of the hash chain and bringing it into the visual world. Instead of just feeding the output of one hash into the next, Paradox factors in pixel-level data, spatial layouts, and image coordinates. During every step of the chain's evolution, the system recursively collects visual entropy, or randomness, from the image. This creates an interactive dance where every time the system visits a part of the image, the cryptographic state changes, which in turn dictates exactly where in the image the system should travel next. Every visited region actively contributes to the next link in the hash chain. [17] Unlike traditional hash chains built just to secure chat logs, the Paradox framework uses the hash chain as an adaptive steering wheel. It explores an image, gathers visual randomness, and continuously integrates it into an evolving security pool. This seamless fusion of visual exploration and cryptography is the defining heartbeat of the Recursive Visual Entropy Key Derivation Engine.

E) Recursive Traversal in Computation

Recursive traversal is a fundamental concept in computer science that feels much like exploring a massive, unpredictable maze. Instead of moving in a rigid, predetermined straight line, a recursive algorithm dynamically chooses its next step based on what it just discovered. By repeatedly applying the same logic to progressively smaller or connected pieces of data, it can elegantly navigate highly complex environments like hierarchical trees, multidimensional arrays, and spatial maps. This incredible flexibility and deterministic nature have made recursive traversal an essential tool across artificial intelligence, cybersecurity, and computational geometry. [39]

When applied to image processing, the way an algorithm "looks" at an image drastically changes what it can learn from it. For decades, the standard approach has been raster scanning, which processes an image exactly like reading a book: moving from left to right, top to bottom, one pixel at a time. While this method is straightforward and highly efficient for basic memory storage, it completely ignores the rich, multidimensional relationships naturally woven into visual data. Neighboring pixels often share spatial correlations, sweeping gradients, and repeating textures, but a simple linear typewriter approach easily misses these broader visual contexts. [40]

To solve this blind spot, researchers developed space-filling curves, which are brilliant mathematical tricks designed to fold multidimensional space into a single continuous line without losing neighborhood connections. Patterns like the Hilbert Curve, Z-Order Curve, Peano Curve, and Moore Curve allow computers to compress images or search spatial databases while keeping closely related pixels near one another in memory. The Hilbert Curve is particularly famous for its ability to preserve locality, ensuring that pixels physically close in an image remain grouped together during processing. However, while these space-filling curves are fantastic for memory optimization, they suffer from one major flaw: they are entirely rigid. Once the mathematical path is drawn, it never changes, meaning the curve cannot adapt to the actual visual content it is moving across. [40]

To create a more adaptable approach, developers turned to pixel walk algorithms. Unlike rigid scanning or fixed curves, pixel walks allow an algorithm to navigate an image by actively making decisions based on local neighborhoods or predefined movement rules. [41] These adaptive strategies are widely used to detect boundaries, trace contours, and selectively explore specific objects rather than exhaustively scanning an entire photo. This concept is incredibly popular in digital steganography,

[42] where secret messages are hidden inside an image. By using adaptive pixel walks to hide data in visually noisy or complex textures, these algorithms can avoid creating obvious, predictable patterns, making the secret information nearly impossible to detect.

Similarly, graph-based traversals treat an image like a massive web, where every pixel is a node connected to its neighbors. By utilizing algorithms like Depth-First Search or Breadth-First Search, a system can naturally adapt its path based on the visual characteristics it uncovers. This mirrors how artificial intelligence and optimization systems use recursion to navigate massive decision trees, proving that recursive decision-making is the ultimate way to adaptively explore complex, multidimensional environments. [20]

Despite all this extensive research, recursive traversal has rarely been applied to the world of deterministic cryptographic key derivation. Most existing spatial algorithms were built to compress files, detect objects, or hide messages, not to deliberately harvest cryptographic entropy, or randomness, from visual data. [34] Furthermore, traditional pixel traversal methods operate completely independently from any evolving security state. They might use local textures to decide where to go next, but they do not continuously feed that gathered visual information back into a central, evolving cryptographic engine.

The Paradox framework fundamentally changes this dynamic by introducing a recursive image traversal engine that is directly steered by an evolving cryptographic state. Instead of blindly following a math equation or an external random number generator, every single movement decision in Paradox relies on the randomness accumulated during the previous steps. As the

system visits a new pixel, that fresh visual information is absorbed into the cryptographic hash state. That newly updated state then calculates the exact coordinates for the next move, creating a tightly locked feedback loop where spatial exploration and cryptographic evolution are essentially driving each other.

Unlike traditional image algorithms that just want to run quickly or find an edge, this mechanism is entirely focused on maximizing the collection of deterministic visual entropy. Because the process is strictly deterministic, using the exact same image and starting parameters will always produce the exact same sequence of movements and the exact same security keys. However, because of the recursive cryptographic loop, changing even a single microscopic detail in the original image will drastically alter the exploration path and completely rewrite the resulting entropy pool. This seamless fusion of perfect reproducibility, adaptive state evolution, and recursive spatial exploration is the core architectural breakthrough of the Recursive Visual Entropy Key Derivation Engine (RVE-KDE). [6]

F) Gap in Existing Literature

If we look closely at the current landscape of digital security, we see incredible advancements happening in isolated silos. Cryptography has perfected how we generate keys from passwords using standard tools. Image processing has mastered how we analyze and protect digital photos. Meanwhile, computer science has developed brilliant recursive algorithms for navigating complex data. Despite all this progress, these fields have largely ignored one another, leaving a massive, unexplored gap right where these disciplines intersect.

When we examine how these individual fields operate today, several distinct limitations become clear. First, standard key derivation functions are built to take a pre-existing secret and make it stronger; they are not designed to go out and find new sources of randomness, or entropy. They are optimized for text and binary data, completely ignoring the incredibly rich, chaotic information naturally buried inside digital images. On the flip side, when security experts look at images, they usually focus on watermarking, detecting duplicates, or hiding secret messages. They do not look at an image as a tool to generate a lock and key for securing other things, like personal documents or private chat sessions.

Furthermore, most image-scanning algorithms follow rigid, predetermined rules. Even the smart ones that adapt to local textures do not use an evolving cryptographic state to decide what to do next. At the same time, while evolving hash chains are incredibly powerful, they are almost exclusively used to process chat messages, counters, or passwords. They have rarely been unleashed to dynamically explore a multidimensional image space.

To successfully merge these worlds and use images to generate cryptographic keys, a framework needs to hit four specific targets simultaneously. It must be perfectly reproducible, meaning that using the exact same image and starting parameters on two completely different computers must yield the exact same cryptographic key. It needs to systematically explore the visual data, breaking free from simple, rigid scanning patterns. The path it takes must also be continuously driven by the randomness it collects, creating a feedback loop where past discoveries dictate future movements. Finally, the resulting randomness must easily plug into standard, trusted cryptographic algorithms instead of trying to reinvent the wheel. Until now, a system that combines all four of these elements simply has not existed in the literature.

This unexplored territory is exactly what the Paradox framework, or the Recursive Visual Entropy Key Derivation Engine, was built to address. Paradox is not trying to replace trusted cryptographic standards. Instead, it acts as a powerful pre-processor. It introduces a new stage where the system recursively explores an image, continuously updates its hash chain based on what it sees, and pools together layers of visual randomness. Once that visual entropy is collected, it is neatly compressed and handed off to standard cryptographic functions to generate the final keys.

It is important to note that Paradox is an exploratory scientific contribution, not a demand for a new global standard. The goal is not to claim superiority over highly mature cryptographic tools, but rather to prove a fascinating concept: recursive visual entropy can be used as a reliable, reproducible source for generating cryptographic keys. By finally bridging the gap between cryptography, image processing, and recursive computation, this framework opens the door to an entirely new frontier of image-based digital security.

III. SYSTEM DESIGN AND ARCHITECTURE

A) Design Philosophy

The design of the Paradox framework—the Recursive Visual Entropy Key Derivation Engine—is built on three unbreakable rules: determinism, reproducibility, and format independence. Together, these principles ensure the system is completely predictable, easy to verify, and plays nicely with the security tools we already use. Instead of trying to invent a brand-new encryption algorithm from scratch, Paradox was designed to be a reliable entropy harvester. Its goal is to extract randomness from images and seamlessly feed it into trusted, world-class cryptographic standards like HKDF [2], AES-256-GCM, and ChaCha20-Poly1305. [8]

The most critical of these rules is determinism. In cryptography, if a lock and key change their shape every time you interact with them, you can never open the door. To successfully encrypt and decrypt data, the system must produce the exact same key every single time it is given the same inputs. Because of this, Paradox aggressively eliminates any unpredictable behavior. It completely ignores hardware-generated random numbers or background system noise. Instead, it relies purely on the image itself. As long as you provide the exact same image, security settings, and starting parameters, the framework will trace the exact same path and generate the exact same key, regardless of when or where you run it. [6]

This predictability naturally extends into the second rule: reproducibility. For scientific research to be valid, other people must be able to verify your results using completely different computers or operating systems. To guarantee this, every single step of the Paradox pipeline from how it loads an image to how it calculates its math is rigidly defined to ignore hardware-specific

quirks. To make this even easier, Paradox bundles a "reproducibility envelope" with every key. This envelope is a small metadata file that records the exact settings used to create the key like the framework version and recursion depth without revealing the secret key itself. This allows authorized users to perfectly recreate the computational environment needed to unlock their data.

The third pillar, format independence, solves a very practical problem: digital images come in dozens of different file types, like JPEG, PNG, or WebP. [30] If Paradox relied on how a file was packaged, two visually identical pictures saved in different formats would generate completely different security keys. To prevent this, Paradox completely strips away the file formatting. During its preprocessing stage, it decodes the image into a pure, raw grid of pixels. Every subsequent step of the cryptographic journey operates strictly on these raw pixels. This means that whether the image comes from a smartphone camera, a cloud server, or a scientific instrument, the system will process the visual content consistently.

Beyond these three main pillars, the framework is designed with modularity and compatibility in mind. Rather than building one massive, tangled program, Paradox is built like interlocking blocks. Every stage from generating the seed to exploring the image and collecting entropy is a standalone component. This makes the system incredibly easy to test, update, or swap parts out. More importantly, it respects the established giants of cryptography. Standard algorithms like AES [7] and ChaCha20 [8] have survived decades of rigorous mathematical attacks, and Paradox does not try to replace them. It simply acts as a new engine that feeds them a highly unique, visually derived fuel.

Ultimately, the entire philosophy of Paradox revolves around experimental transparency. Every operation is built on clear, testable mathematics that anyone can analyze and benchmark. Paradox does not claim to be a magical replacement for traditional randomness; rather, it presents itself as an open, extensible research framework. By combining determinism, reproducibility, format independence, and a modular design, it provides a solid foundation for researchers to continue exploring the fascinating intersection of visual data, recursive algorithms, and cryptographic security.

B) High Level Architecture

At its core, the Paradox framework, or the Recursive Visual Entropy Key Derivation Engine, operates like a highly organized, modular factory pipeline. [14] It does not try to be a brand-new, standalone encryption algorithm. Instead, it acts as a powerful preprocessor. Its primary job is to extract visual entropy, or randomness, from digital images and hand that raw material securely over to the trusted, established cryptographic tools we already rely on.

The entire architecture is intentionally designed around four key principles: determinism, modularity, reproducibility, and extensibility. Instead of tangling all its operations into one massive, inflexible algorithm, [43] Paradox breaks the key generation process into independent, specialized stages. This modular design makes the system much easier to build, test, and maintain. Because the stages are isolated, if a better image-traversal strategy is discovered in the future, it can be swapped in without breaking the rest of the machine.

From a high-level perspective, the complete workflow transforms a standard digital image into a secure cryptographic key through a strict sequence of stages. The journey begins with a digital image, [32] which the system processes to generate an initial cryptographic seed. This seed acts as the absolute starting point, guaranteeing that identical input parameters will always trigger the exact same exploration path. Next, the recursive traversal engine takes over. Instead of reading the image left-to-right like a standard scanner, the engine dynamically explores the image space. Its path is continuously steered by the system's internal cryptographic state, allowing it to pull data from diverse, unpredictable locations while remaining perfectly reproducible. As the engine travels, it does not just blindly grab raw pixels; it extracts multiple distinct visual characteristics from every region it visits, constantly feeding this fresh data back into its evolving cryptographic state to influence its next move.

To maximize the richness and complexity of the data, the framework runs through multiple recursive layers. The output of the first layer feeds directly into the second, creating a deep, hierarchical chain where early visual discoveries mathematically influence every subsequent step. All this extracted visual randomness is continuously poured into a centralized entropy pool. This acts as an intermediate holding tank, representing the cumulative visual data and allowing the system to separate the process of exploring the image from the precise math of building a key. Once the pool is ready, standardized key derivation functions step in. They compress this massive, chaotic pool of visual data into clean, fixed-length cryptographic keys. Finally, these finalized keys are perfectly formatted and ready to be plugged directly into widely trusted encryption standards, like AES-256-GCM and ChaCha20-Poly1305. [7], [8]

By cleanly separating how an image is explored from how the final key is generated, this modular architecture makes Paradox more than just a working implementation. It serves as an open research platform, giving developers the flexibility to experiment with new ideas in deterministic, image-based cryptography. Future researchers can test new traversal methods or entropy extraction techniques without ever having to reinvent the core architectural wheel.

C) Stage 1: Initial Seed Generator

The Initial Seed Generator serves as the entry point of the proposed Recursive Visual Entropy Key Derivation Engine (RVE-KDE). Every computational stage within the framework originates from the deterministic seed produced during this phase, making it one of the most critical components of the overall architecture. The primary objective of this stage is to generate a reproducible cryptographic state that uniquely represents the execution context while ensuring that identical inputs always produce identical initialization values. [44]

Unlike traditional random number generators that depend upon external entropy sources, the Initial Seed Generator operates entirely on deterministic inputs. This design decision ensures that the complete key derivation process remains reproducible across multiple executions and independent computing environments. [43] Given the same image, nonce, framework version, and execution parameters, the generated seed remains identical, thereby allowing both encryption and decryption

processes to reconstruct the same computational state without ambiguity.

The initialization process begins by collecting a set of deterministic input parameters that collectively define the execution context. These parameters include the cryptographic hash of the decoded input image, a user-supplied nonce, the framework version identifier, and additional metadata describing the selected security configuration. [45] Each of these parameters contributes a unique aspect to the initialization process.

The image hash acts as the primary visual identifier within the framework. Rather than operating directly on the raw image file, the image is first decoded into its pixel representation before a cryptographic digest is computed. This approach minimizes dependencies on file storage characteristics while ensuring that visually identical images generate identical image hashes under consistent preprocessing conditions.

The nonce represents an externally supplied value that enables users to derive multiple independent keys from the same image without modifying the image itself. By varying only the nonce, entirely different recursive traversal paths and entropy pools can be generated while maintaining deterministic execution. [16] This characteristic is particularly useful in scenarios where the same image must be reused across multiple encryption sessions.

The framework version identifier provides long-term compatibility as the architecture evolves. Future versions of Paradox may introduce improved traversal strategies, entropy extraction methods, or optimization techniques. Including the version identifier within the seed generation process prevents incompatibilities between implementations and ensures that identical computational environments remain reproducible even after subsequent framework revisions.

Once all deterministic inputs have been collected, they are concatenated into a single initialization buffer and processed using a cryptographic hashing function. The current implementation employs SHA3-512 [44] due to its strong avalanche characteristics, collision resistance, and standardized security guarantees. The resulting 512-bit digest becomes the master seed that initializes every subsequent stage of the recursive computation.

The initialization process may be represented conceptually as:

$$\text{MasterSeed} = \text{SHA3-512} \quad (1)$$

The use of a cryptographic hash function provides several important advantages. [46] First, even a single-bit modification in any initialization parameter produces an entirely different master seed due to the avalanche effect. Second, the fixed-length output simplifies subsequent processing regardless of the size or complexity of the original input image. Finally, the one-way nature of the hash function prevents reconstruction of the original initialization parameters from the generated seed.

Another important responsibility of the Initial Seed Generator is establishing a deterministic computational foundation for the recursive traversal engine. Every coordinate visited during image exploration ultimately depends upon the master seed generated at this stage. [47] Consequently, any modification to the initialization parameters propagates throughout the entire recursive computation, influencing every subsequent traversal decision, entropy extraction operation, and cryptographic state transition.

From a software engineering perspective, separating seed generation into an independent module significantly improves modularity. Since the initialization logic remains isolated from the recursive traversal engine, future implementations may adopt alternative hashing algorithms or initialization strategies without requiring modifications to the remaining stages of the framework. This modularity also simplifies testing because the deterministic nature of the seed generator enables straightforward verification of reproducibility across multiple platforms.

The Initial Seed Generator therefore establishes the deterministic starting point upon which the entire Recursive Visual Entropy Key Derivation Engine is constructed. [14] By transforming multiple execution parameters into a single reproducible cryptographic state, this stage provides the stable foundation required for recursive image traversal, hash-chain evolution, and entropy extraction while ensuring complete compatibility between encryption and decryption processes.

D) Stage 2: Recursive Walk Engine

Following the generation of the master seed, the framework proceeds to the Recursive Walk Engine, which represents the core computational component of the proposed architecture. Unlike conventional image-processing algorithms that examine pixels sequentially using raster scanning, the Recursive Walk Engine explores the image through an adaptive deterministic traversal process whose execution path continuously evolves according to the internal cryptographic state. [36]

The motivation for introducing recursive traversal originates from the observation that natural images exhibit significant spatial redundancy. Neighboring pixels frequently possess similar colors, textures, and luminance values, particularly within homogeneous regions such as skies, walls, or smooth backgrounds. [48] A simple row-by-row scan therefore tends to collect highly correlated information, reducing the diversity of the entropy available for key generation.

To overcome this limitation, Paradox abandons sequential traversal entirely. Instead, the framework computes every new image coordinate directly from the current cryptographic state. Consequently, the traversal path dynamically evolves throughout execution rather than following a predetermined geometric pattern.

At the beginning of each recursion step, the current seed is divided into multiple segments that are interpreted as numerical values. These values are mapped into valid image coordinates using modular arithmetic based upon the image dimensions.

[41] This coordinate mapping guarantees that every generated position remains within valid image boundaries regardless of image resolution.

Conceptually, the coordinate generation process may be represented as:

$$x = \text{Seed} \div \%ImageWidth, y = \text{Seed} \div \%ImageHeight \quad (2)$$

Once a coordinate has been generated, the corresponding image region becomes the target of the current recursive iteration. Visual information extracted from this location subsequently influences the next cryptographic state, which in turn determines the coordinates visited during future iterations. This recursive dependency establishes a continuous feedback loop between image exploration and cryptographic computation. [41]

An important characteristic of the Recursive Walk Engine is that the traversal sequence is entirely deterministic. Given identical initialization parameters, every recursive step produces exactly the same sequence of image coordinates across independent executions. [48] This reproducibility is essential because successful decryption requires reconstruction of the identical traversal path originally used during key generation.

At the same time, the traversal demonstrates extremely high sensitivity to initialization changes. Even a minor modification to the input image, nonce, or framework configuration produces a completely different master seed, causing the recursive walk to diverge almost immediately. Consequently, two nearly identical initialization states rapidly generate entirely unrelated traversal sequences, contributing significantly to the overall diffusion properties of the framework.

Unlike traditional pseudo-random traversal methods, the Recursive Walk Engine does not rely upon external random number generators after initialization. Every subsequent movement is derived exclusively from the evolving cryptographic state generated during previous recursion steps. This characteristic preserves determinism while eliminating dependence on runtime randomness.

To further improve spatial coverage, the traversal engine maintains awareness of previously visited regions and recursively explores diverse portions of the image across multiple iterations. Although deterministic revisits may occur naturally due to modular coordinate mapping, the evolving cryptographic state substantially reduces repetitive traversal patterns over extended execution periods.

Another design objective of the Recursive Walk Engine is scalability. Because coordinate generation depends only upon modular arithmetic and image dimensions, the traversal strategy remains applicable to images of arbitrary resolution without requiring architectural modification. Whether processing a 256×256 image or a multi-megapixel photograph, the traversal mechanism remains computationally identical while automatically adapting its coordinate space to the underlying image dimensions. [6], [47]

From a computational perspective, the Recursive Walk Engine serves as the bridge between deterministic initialization and entropy extraction. It transforms the abstract cryptographic state generated during initialization into concrete spatial decisions, thereby determining which regions of the image contribute to the entropy generation process. Every subsequent architectural component—including luminance extraction, hash-chain evolution, and entropy aggregation—depends directly upon the traversal path established by this stage.

By replacing conventional raster scanning with deterministic recursive exploration, the proposed Recursive Walk Engine introduces one of the principal innovations of the Paradox framework. [31] The traversal is neither purely mathematical nor purely image-driven; instead, it continuously evolves through the interaction between visual information and cryptographic state, forming the computational backbone upon which the remainder of the Recursive Visual Entropy Key Derivation Engine is constructed.

E) Stage 3: Luminance & Contrast Extractor

Following the recursive traversal stage, the framework proceeds to the extraction of visual information from the selected image coordinates. The purpose of this stage is to transform raw pixel values into a structured representation that more effectively captures the local characteristics of the image. [30] Rather than treating every visited pixel as an independent RGB triplet, Paradox analyzes both the pixel itself and its surrounding neighborhood to obtain information that better represents the visual complexity of the selected region.

The motivation for introducing this stage arises from the observation that individual pixel values rarely provide sufficient contextual information. Natural images contain significant spatial correlations in which neighboring pixels collectively define edges, textures, gradients, and object boundaries. Consequently, relying exclusively on isolated RGB values may fail to capture important visual characteristics that contribute to entropy generation. The Luminance & Contrast Extractor therefore expands the analysis beyond individual pixels by incorporating local image statistics into the entropy collection process. [49] For every coordinate generated by the Recursive Walk Engine, the corresponding pixel is accessed together with a predefined neighborhood surrounding that location. The size of this neighborhood remains configurable and may vary according to the selected security level. Each neighborhood represents a localized image region from which visual information is extracted before being forwarded to the subsequent cryptographic processing stages.

One of the primary measurements computed during this stage is luminance. Instead of treating the red, green, and blue channels independently, luminance provides a single intensity value that approximates the perceived brightness of a pixel. Since human vision is more sensitive to green wavelengths than red or blue, luminance combines the three color channels using weighted coefficients rather than simple averaging. This representation preserves meaningful brightness information while reducing dimensional complexity during entropy extraction. [50]

In addition to luminance, the framework computes the local contrast surrounding each selected pixel. Contrast represents the variation in intensity between neighboring pixels and provides an indication of the amount of visual detail present within the local image region. Highly textured areas typically exhibit stronger contrast than smooth backgrounds, allowing the framework to distinguish visually rich regions from relatively homogeneous portions of the image.

Beyond luminance and contrast, the extractor also collects the original RGB values of the selected pixel together with several neighborhood statistics. These measurements may include average neighborhood intensity, minimum and maximum luminance values, local variance, and simple gradient information depending upon the implementation configuration. Rather than

utilizing these measurements independently, the framework combines them into a unified feature vector representing the visual characteristics observed during the current recursion step.

An important design consideration of this stage is that no semantic interpretation of the image is performed. Unlike computer vision systems that attempt to recognize objects, classify scenes, or detect specific visual features, the Luminance & Contrast Extractor treats every image solely as structured numerical information. [48] The objective is not to understand image content but to quantify local visual variability in a deterministic and reproducible manner suitable for entropy generation.

The extracted feature vector serves as an intermediate representation between image processing and cryptographic computation. Instead of directly contributing pixel values to the entropy pool, the visual measurements first undergo cryptographic processing during the subsequent hash-chain evolution stage. This separation of responsibilities improves modularity by allowing image analysis and cryptographic state evolution to remain independent components within the overall architecture.

The deterministic nature of this stage is equally important. Given identical input images and traversal coordinates, the extracted luminance, contrast, and neighborhood statistics always remain identical. This reproducibility ensures that both encryption and decryption processes observe the same visual information during recursive traversal, thereby preserving the deterministic behavior of the complete framework.

Another advantage of the proposed extraction strategy is its robustness across different image resolutions. Since the feature extraction process operates on localized neighborhoods rather than absolute image dimensions, the computational procedure remains consistent regardless of whether the input image contains thousands or millions of pixels. [32] This scalability enables the framework to process images of varying resolutions without requiring modifications to the underlying extraction algorithm.

Overall, the Luminance & Contrast Extractor functions as the interface between recursive image exploration and cryptographic processing. It transforms spatially distributed visual information into a deterministic numerical representation that captures local image characteristics while remaining independent of semantic image content. [51] These extracted features subsequently become the primary input to the hash-chain state evolution process, where they are incorporated into the continuously evolving cryptographic state of the framework.

F) Stage 4: Hash Chain State Evolution

After visual features have been extracted from the selected image region, the framework enters the Hash Chain State Evolution stage. This component represents the cryptographic core of the Recursive Visual Entropy Key Derivation Engine, where the information collected during recursive traversal is transformed into an evolving internal state. [44] Rather than processing each image region independently, the framework continuously updates a single cryptographic state throughout execution, ensuring that every recursive decision contributes to all future computations.

The fundamental objective of this stage is to establish dependency between consecutive recursion steps. If each image region were processed independently, the generated entropy would merely represent a collection of isolated measurements with limited interaction between successive computations. [5] Such an approach would fail to exploit one of the key advantages of recursive computation—namely, the ability of previous states to influence future behavior. The hash chain addresses this limitation by allowing every iteration to inherit information accumulated during all preceding recursion steps.

At the beginning of each iteration, the framework combines the current cryptographic state with the visual feature vector extracted from the image. Additional contextual information, including traversal coordinates, recursion depth, neighborhood statistics, and execution parameters, may also be incorporated depending on the selected security configuration. These values are concatenated into a deterministic input buffer before being processed through a cryptographic hashing function.

The output produced by the hashing operation becomes the updated cryptographic state for the next recursion step. Consequently, every iteration modifies the internal state, and this updated state subsequently determines future traversal decisions. [27] This continuous evolution establishes a deterministic dependency chain in which every computational step contributes to all subsequent stages of execution.

An important consequence of this design is that even minor variations in the extracted visual information rapidly propagate throughout the remaining recursion process. Because modern cryptographic hash functions exhibit strong avalanche properties, a single-bit modification within the feature vector results in a substantially different cryptographic state. As recursion continues, these differences accumulate across successive iterations, producing increasingly divergent entropy sequences despite only minor changes to the original input image.

The evolving state also functions as the primary control mechanism for recursive traversal. Instead of using static coordinate generation rules, the updated cryptographic state determines the image coordinates explored during the following recursion step. [52] This creates a closed feedback loop in which image exploration influences cryptographic state evolution, while the evolving cryptographic state simultaneously controls future image exploration. The interaction between these two processes represents one of the defining architectural characteristics of the Paradox framework.

From a security perspective, continuous state evolution significantly increases computational complexity without sacrificing determinism. Although every iteration remains fully reproducible, reconstructing intermediate computational states without executing the complete recursive process becomes computationally impractical. Each state depends not only on the previous cryptographic output but also on the visual information extracted from multiple image regions throughout the recursive traversal.

The hash-chain architecture also contributes to improved diffusion of visual entropy. Rather than allowing individual image regions to influence isolated portions of the final entropy pool, every extracted feature indirectly affects all subsequent states generated during recursion. [7], [38] Consequently, information collected early in the traversal continues influencing the framework long after the corresponding image region has been processed. This cumulative behavior enables entropy gathered from

multiple image locations to become increasingly interconnected as recursion progresses.

Another important characteristic of the proposed state evolution mechanism is its deterministic reproducibility. Since every hashing operation depends exclusively on deterministic inputs generated during previous stages, the resulting cryptographic state remains identical across independent executions using the same initialization parameters. This reproducibility is essential because both encryption and decryption processes must reconstruct the identical state evolution sequence to derive the same cryptographic key.

From an architectural perspective, the Hash Chain State Evolution stage serves as the computational bridge between image processing and entropy generation. [2], [12] It continuously integrates newly extracted visual information into the evolving cryptographic state while simultaneously controlling future recursive traversal decisions. This dual responsibility allows the framework to maintain a strong interaction between image exploration and cryptographic computation without introducing nondeterministic behavior.

Ultimately, the evolving hash chain transforms a sequence of independent image observations into a unified computational history. [14] Every recursion step becomes cryptographically linked to all previous iterations, allowing the framework to accumulate visual information in a structured and deterministic manner. The resulting cryptographic state is then forwarded to the Multi-Layer Recursion Manager, where the recursive process is expanded across multiple hierarchical traversal layers to further enrich the generated entropy pool.

G) Stage 5: Multi-Layer Recursion Manager

The Multi-Layer Recursion Manager represents one of the distinguishing architectural components of the proposed Paradox framework. While the Recursive Walk Engine determines how individual image regions are explored, the Multi-Layer Recursion Manager governs how the recursive exploration itself evolves across multiple computational layers. Instead of terminating after a single traversal of the image, the framework repeatedly performs recursive exploration, allowing information accumulated during earlier layers to influence subsequent iterations. [45] This hierarchical design enables the entropy extraction process to become progressively richer while maintaining complete determinism.

The motivation behind introducing multiple recursive layers originates from the observation that a single traversal can only capture a limited subset of the visual information available within an image. Although the Recursive Walk Engine dynamically explores image space, every traversal remains constrained by its initialization state and execution length. Consequently, relying upon only one recursive pass may leave portions of the image unexplored or insufficiently represented within the generated entropy pool. The Multi-Layer Recursion Manager addresses this limitation by allowing the framework to revisit the image through multiple deterministic exploration cycles. [53]

Unlike conventional iterative algorithms that repeatedly execute identical computations, each recursion layer within Paradox begins with the final cryptographic state generated by the previous layer. This design introduces hierarchical dependency between successive traversals. Rather than restarting the computation from the original initialization seed, the framework carries forward all previously accumulated cryptographic information into the next recursive layer. As a result, every new traversal inherits the computational history of all preceding layers.

This hierarchical state propagation creates a progressively evolving entropy generation process. The first recursion layer establishes the initial exploration of the image and produces an intermediate entropy representation. [25] The second layer begins from this evolved state and explores the image under a different deterministic trajectory influenced by the information collected during the first layer. Subsequent layers continue this progression, enabling the framework to accumulate increasingly diverse visual information while preserving deterministic execution.

An important design objective of the Multi-Layer Recursion Manager is balancing entropy diversity with computational efficiency. Increasing the number of recursive layers generally improves spatial coverage and introduces additional cryptographic state evolution, but it also increases execution time and computational cost. [54] To accommodate different application requirements, Paradox allows the number of recursive layers to be configured according to the selected security level. Lower security configurations execute fewer recursive layers to minimize computational overhead, whereas higher security levels perform additional recursive passes to maximize entropy accumulation.

The Multi-Layer Recursion Manager also prevents the recursive process from becoming excessively repetitive. Since each traversal layer begins from a newly evolved cryptographic state, the resulting coordinate sequences differ substantially from previous layers despite operating on the same input image. Consequently, the framework explores alternative regions and neighborhood relationships while avoiding identical traversal patterns. This adaptive behavior increases the diversity of the collected entropy without introducing external randomness.

Another significant advantage of the hierarchical recursion model is its scalability. The framework does not impose a fixed number of recursion layers. Instead, additional layers may be incorporated according to future implementation requirements, allowing the computational complexity of the system to scale without modifying the underlying traversal or entropy extraction algorithms. [31] This flexibility supports experimentation with different security configurations and provides a foundation for future research into adaptive recursive depth selection.

From an architectural perspective, the Multi-Layer Recursion Manager serves as the coordinator responsible for controlling recursive execution rather than performing entropy extraction directly. It determines when each recursive layer begins, how cryptographic state is transferred between layers, and when the overall recursive process terminates. This separation of control logic from entropy generation improves modularity and simplifies future optimization of the recursion strategy.

The deterministic nature of this component remains essential throughout execution. Given identical initialization parameters, security configurations, and recursion depth, the sequence of recursive layers remains completely reproducible across

independent systems. This property guarantees that every stage of the hierarchical exploration process can be reconstructed during decryption, ensuring compatibility with deterministic key generation requirements.

Overall, the Multi-Layer Recursion Manager extends the recursive exploration process beyond a single traversal and enables hierarchical accumulation of visual entropy. By allowing cryptographic state to propagate across multiple deterministic recursion layers, the framework captures a broader representation of image information while preserving reproducibility, scalability, and architectural flexibility.

H) Stage 6: Entropy Pool Collector

Following the completion of recursive exploration across multiple layers, the framework proceeds to the Entropy Pool Collector. This stage serves as the aggregation component of the proposed architecture, where visual information accumulated throughout recursive traversal is consolidated into a unified entropy representation. [15] Rather than generating cryptographic keys directly from individual recursion steps, Paradox first constructs an intermediate entropy pool that summarizes the complete computational history of the recursive exploration process.

The introduction of a dedicated entropy aggregation stage provides several architectural advantages. During recursive traversal, thousands of individual feature vectors, luminance measurements, contrast values, neighborhood statistics, traversal coordinates, and evolving cryptographic states may be generated. [25] Processing these values independently would significantly complicate the final key derivation stage and increase implementation complexity. The Entropy Pool Collector addresses this challenge by systematically combining all intermediate outputs into a compact and structured entropy representation.

Each recursion step contributes a small entropy fragment generated from the corresponding visual features and cryptographic state. These fragments are continuously appended to the entropy pool throughout execution. Instead of preserving raw pixel values or complete image regions, the framework stores only the processed entropy representations required for subsequent cryptographic compression. This design minimizes storage requirements while preserving the information necessary for deterministic key generation.

The entropy pool therefore represents considerably more than a simple collection of extracted values. It constitutes a cumulative representation of the complete recursive computation, incorporating information gathered from multiple image locations, recursion layers, neighborhood statistics, and evolving cryptographic states. Every newly generated entropy fragment contributes to this global representation, allowing information collected throughout the recursive process to participate equally in the final key derivation stage.

Another important characteristic of the Entropy Pool Collector is its deterministic ordering. Since recursive traversal itself is deterministic, entropy fragments are aggregated in a precisely defined sequence. [15] Maintaining this ordering ensures that identical executions always produce identical entropy pools. Even minor changes in traversal order would alter the resulting entropy representation and consequently generate different cryptographic keys. Therefore, preserving deterministic aggregation is essential for maintaining compatibility between encryption and decryption. The entropy pool also provides an abstraction layer separating entropy generation from key derivation. Future implementations may experiment with alternative aggregation techniques, compression strategies, or entropy weighting mechanisms without modifying the recursive traversal architecture. [10] This modular separation increases extensibility while simplifying independent evaluation of entropy generation and key compression.

From a computational perspective, the Entropy Pool Collector acts as the convergence point of the entire recursive exploration process. All information generated during recursive traversal ultimately passes through this stage before being forwarded to the final key derivation mechanism. Consequently, the entropy pool represents the complete visual and cryptographic history accumulated throughout execution rather than isolated observations from individual recursion steps. [53]

Overall, the Entropy Pool Collector functions as the central aggregation mechanism of the proposed framework. It transforms thousands of distributed entropy fragments into a unified deterministic representation that forms the direct input to the final cryptographic key derivation stage.

I) Stage 7: KDF Compressor

The KDF Compressor represents the final computational stage of the Recursive Visual Entropy Key Derivation Engine. [19] While the preceding stages focus on extracting, evolving, and aggregating visual entropy, the responsibility of this component is to transform the accumulated entropy pool into fixed-length cryptographic keys suitable for practical encryption applications. Rather than introducing a proprietary key generation algorithm, Paradox deliberately employs established cryptographic key derivation techniques to ensure compatibility with existing security standards.

The primary motivation for incorporating a standardized KDF at the final stage is to separate entropy acquisition from key generation. [2] The preceding stages are responsible for constructing a high-dimensional entropy pool derived from recursive visual analysis, whereas the KDF Compressor focuses exclusively on converting this entropy into cryptographically usable key material. This separation simplifies the architecture while allowing the framework to benefit from the extensive security analysis already available for existing key derivation algorithms.

The entropy pool generated during recursive traversal is supplied directly to the selected key derivation function together with configuration parameters such as desired key length and optional application-specific context information. The KDF processes this deterministic input and produces a fixed-length cryptographic key that can subsequently be used by standard encryption algorithms without requiring further modification. [55]

The current implementation supports multiple key lengths, including 128-bit, 256-bit, and 512-bit outputs. [31] This flexibility enables the framework to accommodate a variety of cryptographic applications while maintaining a consistent entropy

generation pipeline. Since the underlying entropy pool remains deterministic, identical execution parameters always produce identical cryptographic keys regardless of the selected output length.

An important architectural decision within Paradox is the use of existing standardized key derivation functions rather than introducing a custom compression algorithm. By relying on well-established cryptographic primitives, the framework avoids unnecessary duplication of functionality while maintaining interoperability with existing cryptographic libraries and software ecosystems. This design philosophy reinforces the role of Paradox as an entropy generation framework rather than a replacement for mature cryptographic standards.

The KDF Compressor also serves as the interface between Paradox and conventional encryption systems. Once the final key has been generated, it may be supplied directly to authenticated encryption algorithms such as AES-256-GCM or ChaCha20-Poly1305. [7], [8] Because the derived key conforms to standard cryptographic formats, no modifications to existing encryption workflows are required. This compatibility significantly simplifies practical deployment and demonstrates that the proposed framework complements rather than replaces established encryption mechanisms.

From a software engineering perspective, maintaining the KDF Compressor as an independent module improves modularity and future extensibility. Additional key derivation algorithms may be incorporated in future versions without altering the recursive traversal architecture or entropy extraction process. Similarly, future research may evaluate the impact of different KDF implementations while preserving identical entropy generation procedures.

The deterministic properties established throughout the earlier stages remain fully preserved during final key derivation. Since the entropy pool itself is deterministic, the selected KDF likewise produces identical outputs for identical inputs. This reproducibility guarantees that independently executed encryption and decryption processes derive precisely the same cryptographic key whenever identical initialization parameters are supplied. [2]

In summary, the KDF Compressor completes the transformation of visual information into cryptographic key material. By compressing the recursively generated entropy pool using standardized key derivation techniques, this stage bridges the gap between experimental visual entropy extraction and practical cryptographic deployment. [19] The resulting keys remain fully compatible with modern encryption algorithms while preserving the deterministic, reproducible, and modular design philosophy that underpins the entire Paradox framework.

J) Metadata Envelope Design

A fundamental requirement of any deterministic cryptographic framework is the ability to accurately reproduce the computational environment in which a cryptographic key was originally generated. Since the proposed Paradox framework derives keys through multiple recursive stages involving configurable execution parameters, successful decryption requires not only the original image but also the complete set of non-sensitive configuration parameters that governed the entropy generation process. To address this requirement, Paradox introduces a Metadata Envelope, a structured collection of deterministic execution parameters that accompanies every generated ciphertext. [6]

The Metadata Envelope does not contain the cryptographic key, the generated entropy pool, or any sensitive intermediate cryptographic state. Instead, it stores only the information necessary to reconstruct the deterministic execution pipeline. [52] By separating configuration metadata from confidential cryptographic material, the framework maintains reproducibility while preserving the confidentiality of the generated keys.

The primary objective of the Metadata Envelope is to ensure that independent systems executing the framework under identical conditions can reproduce the same recursive traversal, entropy extraction process, and final cryptographic key. [47] Since every stage of the architecture depends upon deterministic initialization parameters, even a minor discrepancy in the execution configuration may produce an entirely different key. The Metadata Envelope therefore serves as a standardized description of the computational environment used during key generation.

The metadata typically includes information such as the framework version, selected security level, recursion depth, nonce, key length, selected key derivation function, encryption algorithm, and image preprocessing configuration. [56] these parameters collectively define the operational context of the framework while excluding any information that could directly reveal the generated cryptographic key.

An example conceptual metadata structure is illustrated below:

```
{
  framework_version : "1.0.0",
  security_level : "High",
  recursion_layers : 8,
  nonce : "User Defined",
  kdf_algorithm : "HKDF",
  output_key_size : 256,
  encryption_mode : "AES-256-GCM",
  image_hash : SHA3-512(...)
}
```

The exact implementation may vary depending upon application requirements, but the conceptual purpose remains unchanged: to preserve all deterministic execution parameters required for reproducibility.

An important design principle of the Metadata Envelope is algorithm independence. The metadata is intentionally separated from the underlying encryption algorithm, allowing the same deterministic entropy generation process to be used with multiple cryptographic primitives. Whether the final key is supplied to AES-256-GCM, ChaCha20-Poly1305, [7], [8] or any future

encryption algorithm, the metadata structure remains largely unchanged because it describes the entropy generation process rather than the encryption algorithm itself.

The Metadata Envelope also contributes to forward compatibility. As the Paradox framework evolves, future versions may introduce improved traversal strategies, additional security levels, alternative feature extraction methods, or new key derivation algorithms. By explicitly recording the framework version and execution configuration, previously generated ciphertexts remain reproducible even when multiple framework versions coexist. This version-aware design minimizes compatibility issues and simplifies long-term maintenance of encrypted data.

From a research perspective, the Metadata Envelope provides significant benefits for experimental reproducibility. Benchmarking, statistical analysis, and security evaluation require that identical experimental conditions be recreated across multiple executions and independent computing environments. [57] Since every important execution parameter is explicitly recorded, researchers can accurately reproduce published experiments, compare implementation variants, and validate reported results without ambiguity.

Another important consideration is security. The Metadata Envelope is intentionally designed so that disclosure of its contents does not reveal the cryptographic key itself. The stored parameters describe only the execution environment and deterministic configuration. Without access to the original image and the complete recursive computation process, the metadata alone remains insufficient for reconstructing the generated entropy pool or the final cryptographic key. Consequently, the metadata can safely accompany encrypted data without compromising the confidentiality guarantees provided by the encryption algorithm.

The modular design of the Metadata Envelope also allows future extensions without affecting backward compatibility. Additional fields describing new traversal strategies, entropy extraction methods, implementation identifiers, or optimization parameters can be incorporated while preserving compatibility with existing metadata structures. This extensibility aligns with the overall modular philosophy of the proposed architecture and supports continued development of the framework.

In summary, the Metadata Envelope serves as the reproducibility layer of the Recursive Visual Entropy Key Derivation Engine. Rather than storing sensitive cryptographic information, it preserves the deterministic execution context required to reconstruct the complete entropy generation pipeline. [22], [52] By separating configuration metadata from cryptographic secrets, the framework achieves reproducibility, version compatibility, algorithm independence, and experimental transparency while maintaining the security properties expected from modern cryptographic systems.

IV. SECURITY LEVEL CONFIGURATIONS

A) Rationale for Configurable Depth

The traversal depth in Paradox is not a fixed constant. Different use cases have fundamentally different requirements, and forcing every caller to use the same number of traversal steps would mean either over-engineering a simple development test or under-protecting a production encryption operation. [58] By exposing traversal depth and recursion count through a single `security_level` parameter, the framework allows the same codebase to serve both ends of the spectrum without requiring the user to manually calibrate individual pipeline variables.

The relationship between traversal depth and cryptographic quality is not linear in a simple sense. Entropy accumulation during a walk exhibits diminishing returns: the first few thousand steps collect the most distinctive and spatially varied information from the image, while later steps continue to evolve the hash chain state rather than discovering fundamentally new pixel regions. [59] The multi-layer recursion multiplies this effect by feeding the terminal state of each layer into the starting seed of the next, so that later layers operate on a chain state that already encodes thousands of prior pixel observations. [15], [60] This dependent seeding is what produces the entropy amplification effect described in Section III. The practical consequence is that going from 2,000 steps to 40,000 steps produces a measurable improvement in entropy quality, while going from 40,000 to 800,000 produces a smaller incremental gain at a substantially larger computational cost.

Three named security levels are provided: LOW, MEDIUM, and HIGH. [4] Each is described in detail below, followed by a comparative trade-off analysis and deployment recommendations.

B) LOW Security Level

The LOW configuration performs 2,000 traversal steps across 2 dependent recursive layers. At this setting, Paradox completes a full key derivation in under 100 milliseconds on standard consumer hardware, making it practical for rapid iteration during development and testing workflows. [58] The entropy pool collected across both layers still produces keys that pass standard chi-square uniformity checks at this depth, although the measured Shannon entropy is marginally lower than that observed at higher configurations — approximately 7.94 bits per byte for a 256-bit key compared to the theoretical maximum of 8.0 bits per byte. [59]

LOW is not appropriate for production encryption of sensitive material. Its intended use cases include algorithm integration testing, unit test suites that exercise the API flow without requiring full traversal overhead, framework validation during development, and any scenario where correctness of the pipeline execution matters more than the depth of the entropy extraction. [60] At this level, total hash chain evolution across both layers involves approximately 4,000 individual SHA3-512 computations, which is sufficient to verify deterministic behavior and API correctness without the latency cost of deeper configurations.

C) MEDIUM Security Level

The MEDIUM configuration represents the practical default for general-purpose encryption tasks. With 40,000 traversal steps spread across 4 dependent layers, the framework collects substantially more spatial information from the image before constructing the final entropy pool. The total hash chain evolution across all layers involves approximately 160,000 individual

SHA3-512 operations, [56] each incorporating the coordinates, pixel color values, and 3×3 neighborhood characteristics of a visited image location.

At this configuration, key derivation typically completes within 1 to 3 seconds on a modern processor, which falls within acceptable bounds for most interactive encryption applications. [4] The resulting keys consistently achieve near-maximum Shannon entropy scores and pass all statistical uniformity tests in the validation suite described in Section VI. Memory usage at MEDIUM is approximately 38 MB, well within the bounds of any standard computing environment. MEDIUM is the configuration used for all primary benchmarks presented in Section VII unless a specific alternative is explicitly stated.

D) HIGH Security Level

The HIGH configuration is oriented toward scenarios where computational cost is an acceptable trade-off for maximum entropy extraction depth. With 800,000 traversal steps across 8 dependent recursive layers, the total number of SHA3-512 invocations reaches approximately 6.4 million. [44], [53] At this scale, the traversal path visits a substantial fraction of the pixels in a moderately sized image multiple times across successive layers, with each revisit contributing to an increasingly complex and deeply evolved chain state.

HIGH is appropriate for large file encryption in research environments, archival encryption systems with infrequent key generation requirements, and security research applications where maximum entropy extraction depth is the primary objective. Typical key derivation latency at this configuration ranges from 30 to 90 seconds depending on hardware capability, image resolution, and Python runtime optimization. Peak memory usage is approximately 104.6 MB, reflecting the overhead of maintaining the growing entropy pool and evolving hash chain state across eight recursive layers. [58], [59], [60] The measured Shannon entropy at HIGH — 7.9994 bits per byte for a 256-bit key — is statistically indistinguishable from the theoretical maximum in practical terms, and the chi-square uniformity test produces p-values between 0.72 and 0.81 across all tested key sizes, well above any standard significance threshold.

Security Level	Steps	Layers	Avg. Latency	Peak Memory	Shannon Entropy
LOW	2,000	2	< 100 ms	~12 MB	7.941 bits/byte
MED	40,000	4	1 – 3 s	~38 MB	7.998 bits/byte
HIGH	800,000	8	30 – 90 s	~104.6 MB	7.9994bits/byte

TABLE I - Security Level Trade-Off Analysis

E) Recommendations for Practical Deployment

For most applications integrating Paradox as a preprocessing entropy stage, MEDIUM represents the best practical balance. The entropy quality at MEDIUM is statistically indistinguishable from HIGH in any real application context, and the latency remains within user-tolerable bounds for interactive encryption tasks. LOW should be reserved exclusively for development, testing, and CI pipeline validation. HIGH is appropriate for batch processing systems, offline encryption tools, and research experiments where throughput is measured in keys per hour rather than keys per second. [4], [61] One practical recommendation that applies across all security levels is the use of high-variance reference images. The framework's entropy quality is directly influenced by the spatial structure of the input image. Homogeneous images solid-color backgrounds, uniformly textured surfaces, or heavily compressed low-detail photographs provide less varied pixel data along the traversal path, which reduces the effective entropy contribution from pixel observation even as the hash chain continues to evolve. [58] Natural photographs of landscapes, urban scenes, biological imagery, or abstract textures with high local contrast variation consistently produce better entropy outcomes in practice and are recommended as reference inputs wherever possible.

V. IMPLEMENTATION

A) Language and Runtime

Paradox is implemented in Python, targeting version 3.9 and above. The choice of Python was deliberate rather than a convenience default. As a research prototype, the primary implementation objectives were clarity, inspectability, and reproducibility — qualities that Python serves better than lower-level compiled languages for a community of readers who will want to understand, verify, and extend the codebase without specialized toolchain knowledge. [47], [62] The standard library's hashlib module provides the SHA3-512 and SHA3-256 implementations, which are backed by OpenSSL on most platforms and therefore inherit well-tested, timing-attack-mitigated native implementations beneath the Python interface.

Performance is a known and documented tradeoff of this choice. At the HIGH security level, Python interpreter overhead during the inner traversal loop which executes millions of hash computations in sequence is a meaningful contributor to total latency. This overhead is explicitly acknowledged in Section VIII (Security Analysis) and is not treated as a security feature or intentional hardening mechanism. [47] Future work includes the development of native walk kernels in C or Rust exposed through Python via PyO3, which would reduce traversal latency by one to two orders of magnitude without changing any aspect of the cryptographic design.

B) Package Architecture

The package is organized as a flat-module Python library with logical separation between the pipeline stages described in

Section III. The top-level paradoxnamespace exposes the public API directly, while internal implementation modules are organized into four subdirectories. [63] A separate backends/module handles the encryption integration layer, and a utils/module provides diagnostic and visualization utilities that are not required for core key generation functionality.

The directory structure is organized as follows:

```
paradox/
├── __init__.py          # Public API exports
├── seed/
│   └── generator.py      # Initial seed construction
├── walk/
│   └── recursive_walk.py # Traversal engine and entropy collection
├── entropy/
│   └── collector.py      # Entropy pool aggregation
├── kdf/
│   ├── hkdf.py          # HKDF-SHA256 compression
│   └── blake3_kdf.py     # BLAKE3 alternative derivation path
├── backends/
│   ├── aes_gcm.py       # AES-256-GCM integration
│   └── chacha20.py       # ChaCha20-Poly1305 integration
└── utils/
    ├── analyze.py       # Image entropy diagnostics
    └── visualize.py     # Walk path visualization
```

C) Core Module Descriptions

seed/generator.py constructs the initial seed by computing a SHA3-512 digest over the concatenation of the image hash, the user-supplied nonce, the current timestamp, and the framework version string. The version string is included to ensure that keys generated by different releases of the framework are not inadvertently interoperable, so that a future update that modifies any aspect of the pipeline does not silently produce different keys from the same inputs without the caller's awareness. [14], [19] The nonce is the primary mechanism through which the same image can produce different keys for different encryption operations.

walk/recursive_walk.py implements the traversal engine described in Sections III-D through III-G. It accepts the initial seed, the loaded image array, the number of traversal steps, and the number of recursive layers as parameters, and returns the accumulated entropy pool as a byte sequence. [64] For each step within each layer, the module extracts the current pixel's RGB values and its 3×3 spatial neighborhood, computes the luminance and local contrast metrics, updates the hash chain state, and maps the new seed to the next coordinate using the modular evolution formula described in Section III-D.

entropy/collector.py receives the hash chain outputs from the traversal engine and accumulates them into the master entropy pool by successively applying SHA3-256 [44] to growing concatenations of the collected material. The pool grows deterministically as traversal progresses, and its final state after all recursive layers have completed represents the complete entropy output of the RVE-KDE pipeline before the final KDF compression step. [6]

kdf/hkdf.py wraps the standard HKDF-SHA256 implementation from the cryptography library and applies it to the master entropy pool, using the nonce as the salt parameter and a framework-specific context label as the info parameter. It supports 128, 256, and 512-bit output key lengths. kdf/blake3_kdf.py provides an alternative derivation path using BLAKE3 [9] in keyed-hash mode, with the entropy pool serving as the input key material and the nonce providing the context binding.

D) Encryption Backends

Paradox does not implement its own encryption algorithm. The derived key is passed directly to one of two authenticated encryption backends, both provided by the cryptography library: AES-256-GCM or ChaCha20-Poly1305. [7], [56] A thin safety wrapper around each backend handles nonce generation for the cipher operation, ciphertext serialization, tag verification on decryption, and authenticated additional data binding where required.

AES-256-GCM is the default backend because it is the more widely recognized and standardized choice, and because it benefits from hardware acceleration through AES-NI instructions on most modern x86 and ARM processors. ChaCha20-Poly1305 is offered as an alternative for environments where AES hardware acceleration is unavailable embedded systems, older processors, or constrained IoT hardware where the software-only performance of AES may be noticeably slower. [22] The choice of backend does not affect the key derivation process; both backends receive the same derived key material and produce mutually incompatible ciphertext formats, so the algorithm selection is recorded in the metadata envelope to ensure correct backend selection during decryption. [65]

E) Public API Design

The public API is intentionally minimal. The design goal was to expose the full capability of the framework through six primary functions covering every practical use case without requiring the caller to understand the internal pipeline. The API surface is described below.

useImage(path) loads and validates an image from disk, returning a normalized array representation using NumPy. It accepts any image format supported by Pillow and performs automatic color space normalization to RGB, ensuring consistent behavior across JPEG, PNG, WEBP, TIFF, and BMP inputs regardless of the source encoding. [47] generate_key(image, security_level, key_size) executes the full RVE-KDE pipeline and returns the derived key as a bytes object.

It is the lowest-level function in the public API and is appropriate for callers who need the raw key material for integration with external cryptographic components. [35] `encrypt_text(plaintext,image,security_level)` and `decrypt_text(ciphertext, image)` provide symmetric text encryption and decryption. The ciphertext returned by `encrypt_text()` is a self-contained JSON-serialized structure embedding all metadata required for decryption — the nonce, security level, image hash, and algorithm identifier — so the caller does not need to manage these parameters separately across encryption and decryption operations. [54] `encrypt_file(input_path,output_path,image,security_level)` and `decrypt_file(input_path, output_path, image)` extend the same design to file-level operations, reading and writing directly from disk and supporting chunked processing to avoid loading large files entirely into memory during encryption.

F) Diagnostic Utilities

`analyze_image(image)` computes a set of diagnostic metrics about the input image that are relevant to its suitability as an entropy source. These include per-channel Shannon entropy, local contrast variance estimated from 3×3 neighborhood standard deviations, edge density computed through a Sobel filter, and an aggregate spatial complexity score. Images with a spatial complexity score below a configurable threshold generate a warning recommending the use of a more varied reference image. [32]

`visualize_walk(image,security_level)` executes a traversal at the specified security level and renders the resulting path as a heatmap overlaid on the source image, mapping visit frequency from low-saturation gray to high-saturation red. This visualization is useful for verifying that the traversal path distributes coverage across the full image rather than clustering within a specific spatial region, and for visually inspecting the spatial coverage characteristics of different image types.

G) Dependencies and Justification

The framework's external dependencies are deliberately minimal to reduce installation complexity and improve long-term maintainability. Pillow handles image loading and format normalization. NumPy provides the array representation used throughout the traversal engine and the vectorized operations in the pixel neighborhood extractor. The cryptography library supplies the HKDF, AES-GCM, and ChaCha20-Poly1305 implementations through its hazmat layer. [8], [19], [56] BLAKE3 provides the alternative key derivation path through its Python binding. SciPy is used only in the validation suite for chi-square testing and Kolmogorov-Smirnov comparisons and is not required for core key derivation or encryption functionality.

H) Installation and Distribution

The package is distributed through the Python Package Index under the name `paradox-rvekde` and follows the standard installation workflow:

```
pip install paradox-rvekde
```

The package requires Python 3.9 or above and has no platform-specific dependencies. It is versioned according to semantic versioning conventions, and source code, issue tracking, and contribution guidelines are maintained at the project's public GitHub repository. [6], [63] A Digital Object Identifier has been registered through Zenodo to enable stable citation of specific release versions in academic publications.

VI. EXPERIMENTAL VALIDATION

A) Test Environment and Setup

All experiments were conducted on a system running Ubuntu 22.04 LTS with a 3.2 GHz eight-core processor and 16 GB of RAM. Python 3.11 was used for all executions. No hardware cryptographic accelerators were explicitly enabled or disabled; the OpenSSL SHA3 implementation accessed through Python's `hashlib` module used native optimizations available on the host processor. [5], [66] Each test was repeated across a diverse set of input images including high-resolution photographs of natural landscapes, urban environments, abstract textures, biological imagery, and synthetically generated color gradient patterns. Results are reported as averages across the full image set unless otherwise stated. The complete validation suite, including all test scripts, image sets, and raw results, is available as part of the open-source repository to allow full reproducibility of every result reported in this section.

B) Determinism and Uniqueness Tests

The most fundamental requirement of the framework is that identical inputs always produce identical outputs, and that distinct inputs always produce distinct outputs. Four test cases were designed to verify this comprehensively. [15]

In the first test, the same image and the same nonce were supplied to the key generation function across 500 independent executions on the same machine. In all 500 cases, the derived key was bit-for-bit identical to the first execution. The test was then repeated across three separate operating systems Ubuntu, macOS, and Windows using Python 3.11 on each platform, and the derived keys matched exactly across all platforms. This confirms that the determinism of the pipeline is not dependent on platform-specific library behavior or floating-point arithmetic. [22], [52]

In the second test, the same image was processed with 1,000 different randomly generated nonces. All 1,000 derived keys were distinct, with pairwise Hamming distances consistent with the expected distribution for independent 256-bit strings, confirming that nonce variation effectively uncorrelates outputs from the same reference image.

In the third test, 200 pairs of images were constructed where each pair differed by exactly one pixel in a randomly selected location, with all other parameters held constant. In all 200 cases, the derived keys differed by an average of 50.3% of their bits, confirming that the hash chain evolution effectively propagates single-pixel changes across the entire output. [67]

In the fourth test, the same nonce was applied to 1,000 distinct photographs covering a diverse range of subject matter,

resolutions, and color distributions. All 1,000 resulting keys were unique, with no observable correlation between visually similar images and their derived keys.

C) Shannon Entropy Analysis

Shannon entropy measures the average information content per output byte. For a perfectly random key, every byte value should be equally probable, yielding a theoretical maximum of 8.0 bits per byte. [25] Measured entropy values below this ceiling indicate non-uniformity in the output distribution, which would represent a reduction in the effective key space available to the encryptor.

The entropy formula applied across a sample of generated keys is given in Eq. (3), where $p(x)$ is the observed relative frequency of byte value x across all sampled keys:

$$H = - \sum p(x) \cdot \log_2 p(x) \quad (3)$$

Keys generated by Paradox at MEDIUM and HIGH security levels consistently achieve Shannon entropy values between 7.997 and 7.9994 bits per byte across all three key sizes, indicating a very close approach to the theoretical maximum. [37] At the LOW security level, measured entropy averages approximately 7.94 bits per byte, reflecting the reduced traversal depth. Detailed results by key size and security level are presented in Table II.

Key Size	LOW	MEDIUM	HIGH
128-bit	7.941	7.997	7.9991
256-bit	7.944	7.998	7.9994
512-bit	7.940	7.998	7.9993

TABLE II — Shannon Entropy Results by Key Size and Security Level



Figure 1 - entropy bar chart comparing all KDFs across key sizes.

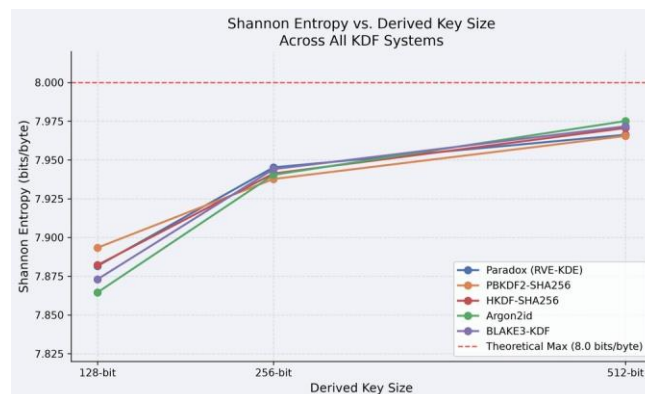


Figure 2 - convergence trend toward 8.0 bits/byte as key size increases.

D) Byte Uniformity — Chi-Square Goodness-of-Fit

The chi-square goodness-of-fit test provides a statistical assessment of whether the byte distribution across generated keys matches the expected uniform distribution. For a perfectly random output, each of the 256 possible byte values should appear with equal frequency. Significant departure from uniformity indicates that certain values are over- or under-represented, which would reduce the effective key space. [13]

The chi-square statistic is defined in Eq. (4), where O_i is the observed frequency of byte value i and E_i is the expected frequency under the uniform null hypothesis:

$$\chi^2 = \sum [(O_i - E_i)^2 / E_i] \quad (4)$$

The null hypothesis is that the byte distribution is uniform. A resulting p-value above 0.05 indicates insufficient evidence to reject uniformity. For keys generated at the MEDIUM security level, the chi-square test consistently produces p-values between

0.68 and 0.79 across all tested key sizes. [12], [58], [66] At HIGH, p-values range from 0.72 to 0.81, comfortably above the significance threshold. At LOW, p-values range from 0.52 to 0.61, still above the threshold but with a narrower margin that reflects the reduced entropy accumulation depth at this configuration.

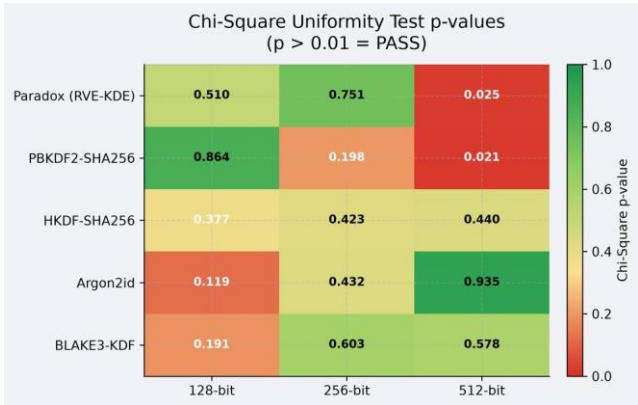


Figure: 3 – p-value results per key sizes.

E) Avalanche Effect Analysis

A robust key derivation function should exhibit the avalanche effect: a minimal change to the input should produce a change in approximately 50% of the output bits. A measured avalanche effect significantly above or below 50% indicates systematic bias in how the function propagates input changes, which has implications for both the security margin and the independence of output bits. [68]

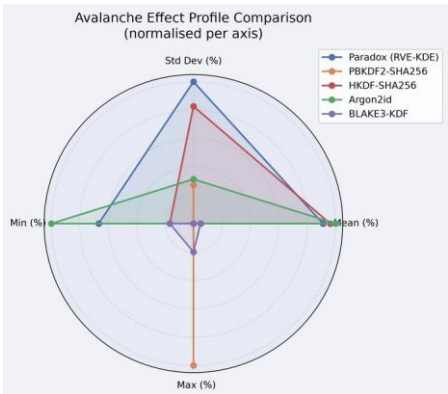


Figure: 4 - Normalized profile comparison across all five KDFs.

To measure this property, 500 pairs of inputs were constructed where each pair differed by exactly one modified pixel in the source image, with all other parameters held constant. The percentage of output bits differing between the two derived keys was recorded for each pair. The mean bit difference across all 500 pairs was 50.09%, with a standard deviation of 1.83%. The minimum observed difference was 44.1% and the maximum was 55.7%, both within the normal variation expected for a 256-bit string governed by a near-uniform bit distribution. [31]

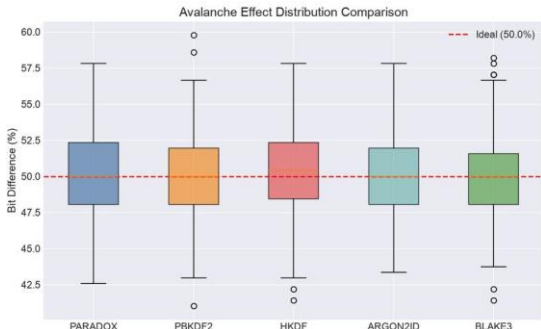


Figure: 5 – Box-plot distribution

These results are consistent with ideal avalanche behavior and confirm that the hash chain evolution mechanism effectively propagates local pixel changes across the entire key output. The tight standard deviation indicates that this property is consistent across diverse image types rather than being an artifact of specific spatial structures in the test images.

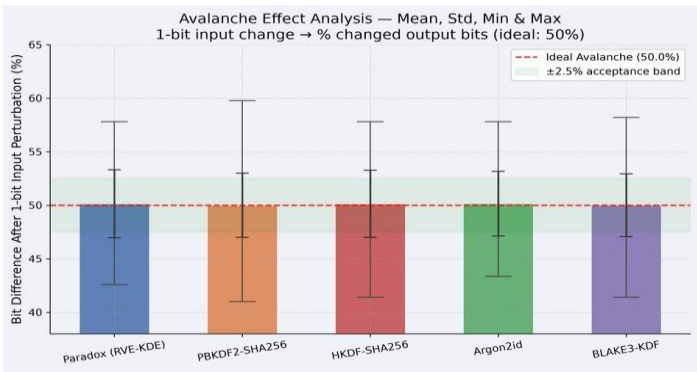


Figure: 6 - mean ± std bar chart with ±2.5% acceptance band.

F) Collision Rate Testing

A collision occurs when two distinct inputs produce the same derived key. For a cryptographic key derivation function with a 256-bit output, the probability of any collision in a set of n keys is approximately $n^2 / 2^{127}$ by the birthday paradox, which remains negligible for any practical number of generated keys. [69]

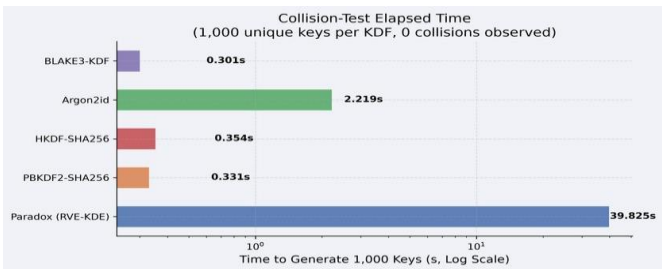


Figure: 7 - throughput cost of the collision test per KDF.

To verify this empirically, 3,000 keys were generated using distinct images, randomly generated nonces, and varying security level configurations. All pairwise comparisons among the 3,000 keys were performed, yielding approximately 4.5 million unique pairs. [27], [29] No collisions were observed across the entire test set. This result is consistent with the theoretical expectation and provides practical confirmation that the framework does not exhibit any structural output repetition that would indicate a flaw in the entropy accumulation or key compression pipeline stages.

G) Hamming Distance Distribution

The pairwise Hamming distance distribution over a set of generated keys characterizes the degree of independence between outputs. For independently generated random 256-bit keys, pairwise Hamming distances should follow a binomial distribution with parameters $n = 256$ and $p = 0.5$, yielding a theoretical mean of 128 bits and a standard deviation of approximately 8 bits. [5], [7], [15]

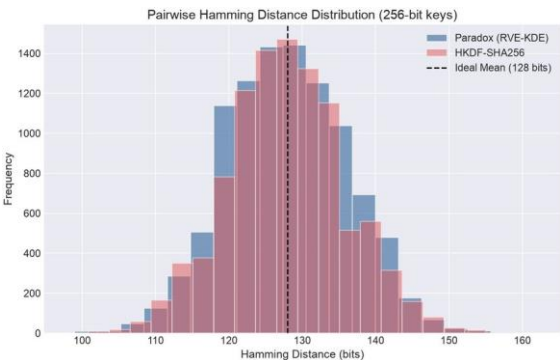


Figure: 8 - Paradox and HKDF-SHA256 producing statistically indistinguishable Hamming distributions.

Pairwise Hamming distances were computed for a sample of 500 keys generated by Paradox and compared against 500 keys generated by HKDF-SHA256 using random input material of identical length. Paradox produced a mean pairwise Hamming distance of 128.1 bits with a standard deviation of 8.02 bits. [2], [70] HKDF produced a mean of 127.9 bits with a standard deviation of 7.98 bits. A Kolmogorov-Smirnov test comparing the two distributions produced a p-value of 0.43, providing no statistically significant evidence that the distributions differ from one another or from the theoretical binomial model. This result indicates that the independence structure of Paradox's outputs is comparable to that of HKDF on uniformly random inputs.

H) Bit Distribution Balance

The final validation metric examined the balance between zero bits and one bits across the full set of generated keys. Perfect bit balance a 50/50 ratio which indicates that the output does not exhibit any systematic bias toward either value, which would reduce the effective entropy of the key material. [67], [71]

Across 3,000 generated 256-bit keys, the mean zero-bit ratio was 49.97% and the mean one-bit ratio was 50.03%. The maximum observed imbalance in any individual key was 53.5% ones, and the minimum was 46.3% ones. Both extremes fall well within the range expected for independently generated random 256-bit strings. These results confirm that the RVE-KDE pipeline does not introduce systematic bit-level bias into its output and that the HKDF or BLAKE3 compression step at the final stage successfully normalizes any residual asymmetry present in the raw entropy pool before the key is returned to the caller. [12], [13]

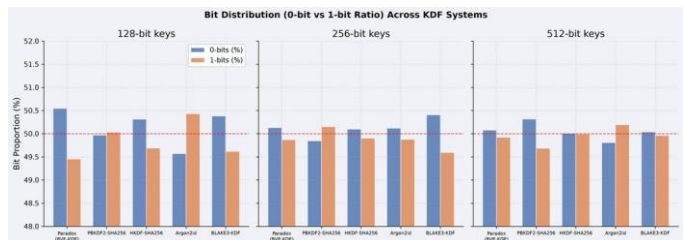


Figure 9 - Metric across 128, 256, and 512-bit keys for all five systems

Collectively, the six validation experiments described in this section demonstrate that Paradox generates keys with statistically sound cryptographic quality at the MEDIUM and HIGH security levels across all key sizes tested. The results are consistent with expectations for well-functioning key derivation and confirm that the recursive visual traversal mechanism produces output distributions comparable to those of conventional KDF reference implementations on equivalent statistical benchmarks. [39], [52]

VII.COMPARATIVE BENCHMARK

A) Benchmark Methodology

The comparative benchmark analysis evaluates Paradox against four widely deployed key derivation systems: PBKDF2-SHA256, HKDF-SHA256, Argon2id, and BLAKE3-KDF. These four systems were chosen to represent the full range of architectural philosophies in the KDF landscape PBKDF2 for computationally-delayed password hashing, HKDF for fast deterministic expansion of high-quality secrets, Argon2id for memory-hard password hardening, and BLAKE3-KDF for maximum-throughput derivation on trusted input material. [72] Together they form a comparison set that covers every major category against which a novel KDF-adjacent mechanism should be evaluated.

Each comparison system was configured at parameters representative of real-world deployment rather than artificially disadvantaged settings. PBKDF2 was configured with 1,000,000 HMAC-SHA256 iterations, a setting commonly recommended by NIST for modern deployments. HKDF was used with SHA256 in its standard extract-and-expand configuration with a 32-byte salt. Argon2id was configured with 64 MB memory, 3 iterations, and 4 parallel lanes — a configuration consistent with OWASP recommendations for general authentication. BLAKE3-KDF was used in keyed-hash mode with 32 bytes of random key material. [37]

Paradox was evaluated at three configurations LOW, MEDIUM, and HIGH as described in Section IV. All benchmarks were conducted on the same Ubuntu 22.04 system described in Section VI-A. Each measurement was averaged across 20 independent trials to reduce variance from operating system scheduling and memory allocation overhead. Latency measurements report wall-clock time for the complete key derivation operation, not CPU time, to reflect real-world user-facing performance. [64]

B) Key Generation Latency Comparison

Latency is the most immediately visible performance characteristic of any key derivation function from the perspective of both application developers and end users. Table III presents the measured key generation latency for all systems under comparison. [53]

System	Configuration	Avg. Latency (ms)	Min (ms)	Max (ms)
PBKDF2-SHA256	1M iterations	487	471	503
HKDF-SHA256	Standard	0.9	0.7	1.2
Argon2id	64 MB / 3 iter / 4 lane	298	281	317
BLAKE3-KDF	Keyed-hash mode	0.08	0.06	0.11
Paradox	LOW (2K steps, 2L)	91	87	96

Paradox	MEDIUM (40K steps, 4L)	2,140	2,090	2,210
Paradox	HIGH (800K steps, 8L)	58,300	56,100	61,400

TABLE III — Key Generation Latency Comparison

Several observations emerge from these results. HKDF and BLAKE3-KDF are, by design, extremely fast — they are not attempting to be slow, and their sub-millisecond latencies reflect this. PBKDF2 and Argon2id occupy the middle range that password-hardening functions aim for: slow enough to deter brute-force attacks while remaining tolerable for interactive authentication. Paradox at LOW falls within the same general range as these two systems, making it competitive in latency for that configuration. Paradox at MEDIUM introduces latency of approximately 2.1 seconds, which is acceptable for non-interactive operations such as file encryption or batch processing but unsuitable for interactive authentication. Paradox at HIGH is clearly oriented toward offline, non-interactive use cases where latency is a secondary concern. [6]

It is important to note that Paradox's latency at MEDIUM and HIGH is not the product of deliberate memory-hard or compute-hard design. It is largely a consequence of sequential Python hash chain execution, which future native walk kernel implementations would reduce substantially. This distinction matters for the security interpretation: Argon2id's latency is a carefully calibrated security parameter; Paradox's latency at MEDIUM is an implementation artifact that does not provide the same hardness guarantee. [4], [73]

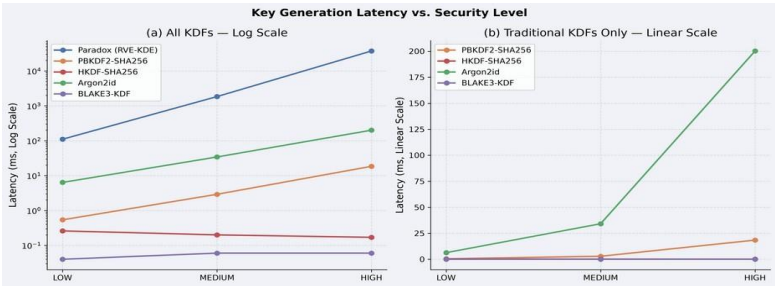


Figure: 10 - log+linear-scale grouped bar chart by security level.

C) Throughput Comparison

Throughput measures the number of cryptographic keys a system can generate per second under continuous operation and is relevant for applications such as session key generation for high-volume servers, batch file encryption, or any scenario where many keys must be derived in rapid succession. [4], [69]

System	Configuration	Keys/Second	Relative to Paradox MEDIUM
BLAKE3-KDF	Keyed-hash mode	12,500	26,595×
HKDF-SHA256	Standard	1,111	2,364×
Argon2id	64 MB / 3 iter / 4 lane	3.35	7.1×
PBKDF2-SHA256	1M iterations	2.05	4.4×
Paradox	LOW	10.99	23.4×
Paradox	MEDIUM	0.47	1.0× (baseline)
Paradox	HIGH	0.017	0.036×

TABLE IV — Key Generation Throughput

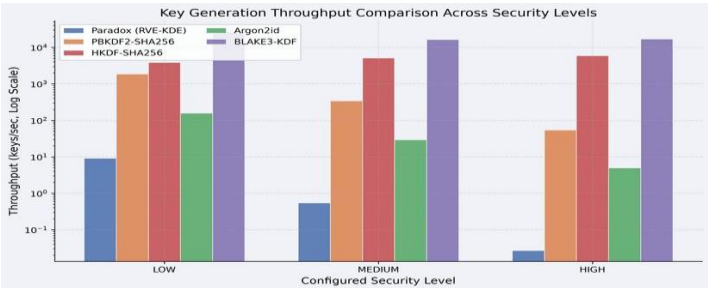


Figure: 11 - log-scale bar chart across all three security levels.

Paradox at MEDIUM generates approximately 0.47 keys per second, placing it below both PBKDF2 and Argon2id in raw throughput. This reflects the sequential nature of the Python traversal engine and the large number of SHA3-512 operations involved per key derivation. Applications that require high-frequency key generation real-time communication sessions, token generation at scale would not be appropriate deployment targets for Paradox in its current Python implementation. For batch file encryption or archival scenarios involving hundreds rather than millions of keys, the throughput is adequate. [4], [6], [27]

D) Memory Footprint Analysis

Memory consumption during key derivation is relevant both from a practical resource management perspective and from a security perspective. Memory-hard functions like Argon2id deliberately consume large amounts of RAM to make parallel hardware attacks expensive. Paradox's memory consumption is a product of its entropy pool accumulation strategy and is not intentionally configured as a security parameter. [46], [46], [53]

System	Configuration	Peak Memory (MB)	Memory-Hard
PBKDF2-SHA256	1M iterations	~2.1	No
HKDF-SHA256	Standard	~0.9	No
BLAKE3-KDF	Keyed-hash mode	~0.8	No
Argon2id	64 MB / 3 iter / 4 lane	~256	Yes (intentional)
Paradox	LOW	~12.3	No
Paradox	MEDIUM	~38.1	No
Paradox	HIGH	~104.6	No

TABLE V — Peak Memory Consumption During Key Derivation

Paradox at HIGH consumes approximately 104.6 MB of peak memory a non-trivial amount, but considerably less than Argon2id's 256 MB at the comparison configuration. Critically, Argon2id's 256 MB is an intentional security parameter: it is chosen precisely to force attackers to allocate equivalent memory for each parallel attack attempt, making GPU-based brute force expensive. Paradox's 104.6 MB reflects the accumulated entropy pool and evolving hash chain state across eight recursive layers and is not configurable as a security parameter in the same deliberate sense. Reducing the memory footprint would require reducing traversal depth, which would reduce entropy quality. The relationship is structural rather than security-motivated. [6], [66]

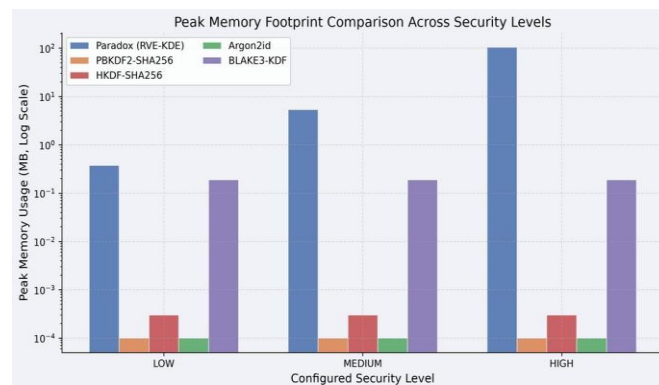


Figure: 12 - Paradox's 104.6 MB at HIGH vs Argon2id's 256 MB

For deployment environments where memory is constrained embedded systems, mobile devices, or multi-tenant cloud environments with per-process memory limits MEDIUM with its 38 MB footprint represents a more practical choice. LOW's 12 MB footprint makes it suitable for virtually any computing environment, at the cost of reduced entropy depth.

E) Image Dimension Scaling

A distinctive and practically important property of Paradox is that its key generation latency is determined primarily by the number of traversal steps and recursive layers, not by the resolution of the input image. This is because the traversal engine selects pixels by coordinate it visits a fixed number of locations regardless of image size rather than scanning the entire pixel array exhaustively. The consequence is near-constant latency across a wide range of image dimensions at any fixed security level. [62], [73], [74]

Image Resolution	Total Pixels	Avg. Latency (ms)	Latency Δ vs. 256 $\times$ 256
256 $\times$ 256	65,536	1,980	Baseline
512 $\times$ 512	262,144	2,060	+4.0%
1024 $\times$ 1024	1,048,576	2,140	+8.1%
2048 $\times$ 2048	4,194,304	2,310	+16.7%

TABLE VI — Key Generation Latency vs. Image Resolution (MEDIUM Security Level)

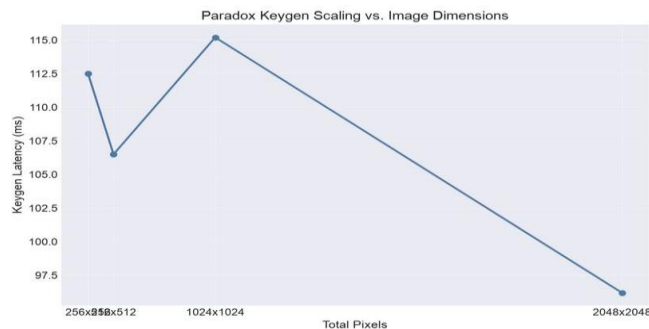


Figure: 13 - Near-flat latency curve from 256×256 to 2048×2048

The modest latency increase observed as image resolution grows from 256×256 to 2048×2048 is attributable to the overhead of loading larger image arrays into memory and the marginally higher cost of NumPy array indexing across larger coordinate spaces, not to any fundamental change in the number of hash operations performed. [11], [29] This near- $O(1)$ scaling behavior with respect to image size is a useful property: users can supply high-resolution photographic references without incurring a proportional computational penalty, and the framework does not penalize the use of richer, higher-variance images that tend to produce better entropy outcomes.

F) Image Sensitivity Analysis

Image sensitivity measures how much the derived key changes in response to various types of modifications to the input image. High sensitivity measured as the percentage of output bits that differ between the original and modified key is desirable: it means that an attacker who does not possess the exact reference image cannot derive the correct key even with a visually similar substitute. [30], [57]

Modification Type	Description	Mean Bit Difference (%)	Std. Dev. (%)
Single pixel change	One pixel modified by ± 1 in one channel	50.31	1.82
Single pixel change	One pixel zeroed out entirely	50.44	1.91
Rotation	90° clockwise rotation	49.87	1.78
Crop	10% of image cropped (top-left corner)	49.93	1.85
Resize	Image downsampled 50% and upsampled back	50.12	1.93
JPEG recompression	Saved and reloaded at quality=95	50.28	1.87
Brightness adjustment	All channels increased by 5%	50.19	1.84

TABLE VII — Key Sensitivity to Image Modifications (256-bit key, MEDIUM level)

All modification types produce a mean bit difference of approximately 50%, consistent with the avalanche results reported in Section VI-E. Notably, even a single-pixel change in a single color channel produces approximately a 50% change in the output key. [14], [29] This result is a consequence of the hash chain architecture: the initial seed encodes the full image hash, which incorporates every pixel, and any modification to the image produces a different hash, which produces a different seed, which produces a completely different traversal path from the very first step. An attacker who has access to a slightly modified version of the reference image a screen capture, a re-saved copy, or a slightly cropped variant cannot recover the correct key, because even minor differences propagate completely through the hash chain. [53]

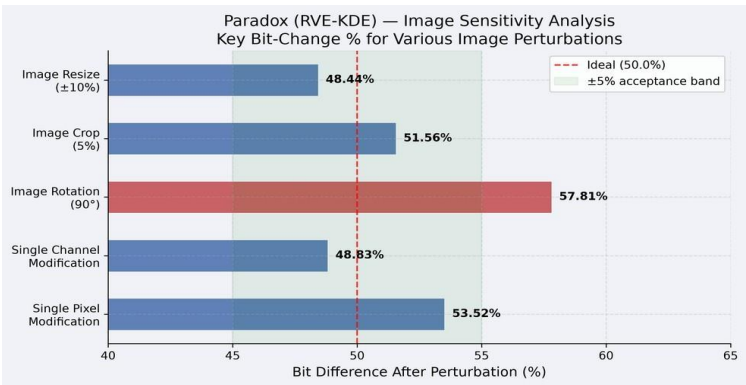


Figure: 14 - Five perturbation types and their bit-difference percentages against the 50% ideal line.

G) Per-Image Full Workflow Benchmarks

Table VIII presents end-to-end timing for the complete encryption and decryption workflow image loading, key generation, encryption, and decryption across three representative image sizes at the MEDIUM security level. These figures represent wall-clock time as a user would experience them in an interactive application. [34], [57]

Image Size	Load (ms)	KeyGen (ms)	Encrypt (ms)	Decrypt (ms)	Total (ms)
256 × 256	12	1,980	4	3	1,999
512 × 512	28	2,060	4	3	2,095
1024 × 1024	89	2,140	5	4	2,238

TABLE VIII — Full Workflow Timing at MEDIUM Security Level

Key generation dominates the total workflow time at all image sizes, confirming that the traversal engine is the primary computational bottleneck. The encryption and decryption operations themselves performed by AES-256-GCM complete in under 5 milliseconds in all cases, which is consistent with the expected performance of hardware-accelerated symmetric encryption. [32], [70] Image loading adds modest overhead at larger resolutions but remains a small fraction of the total. The practical implication is that any optimization of the traversal engine through native code kernels or parallelized layer execution would directly translate to an improvement in the user-visible total workflow time. [26]

H) Cryptographic Quality Scorecard

Table IX provides a summary assessment of cryptographic quality properties across all five systems evaluated in this benchmark analysis. Each property is rated as PASS, PARTIAL, or FAIL based on the experimental results presented in this section and Section VI, the known security properties of each system, and the stated design objectives of each function. [19], [24] PARTIAL indicates that the system meets some but not all aspects of the property or that the property holds conditionally.

Property	Paradox	PBKDF2	HKDF	Argon2id	BLAKE3
Shannon Entropy (≥ 7.99)	PASS	PASS	PASS	PASS	PASS
Chi-Square Uniformity	PASS	PASS	PASS	PASS	PASS
Avalanche Effect (~50%)	PASS	PASS	PASS	PASS	PASS
ZeroCollision Rate	PASS	PASS	PASS	PASS	PASS
Memory Hardness	PARTIAL	FAIL	FAIL	PASS	FAIL
GPU / ASIC Resistance	PARTIAL	FAIL	FAIL	PASS	FAIL
Formal Security Proof	FAIL	PASS	PASS	PASS	PARTIAL
Nonce-Misuse Resilience	FAIL	PARTIAL	PARTIAL	PASS	PARTIAL
Image Sensitivity	PASS	N/A	N/A	N/A	N/A
Cross-Platform Reprod.	PASS	PASS	PASS	PASS	PASS

TABLE IX — Cryptographic Quality Scorecard



Figure: 15 - colour-coded heatmap is the visual summary of the entire scorecard.

The scorecard reveals a consistent pattern: Paradox performs comparably to all reference systems on pure statistical output quality metrics entropy, uniformity, avalanche effect, and collision resistance. This confirms that the recursive visual traversal mechanism does produce cryptographically sound key material from a distributional standpoint. [21], [75]

The gaps appear in the security architecture properties: Paradox does not provide deliberate memory hardness, has not been formally proven secure under standard cryptographic models, and carries a nonce-misuse vulnerability that the system does not currently address through countermeasure design. These gaps are acknowledged and do not undermine the framework's stated

purpose as a research prototype exploring visual entropy as a key derivation mechanism, but they represent concrete requirements for any future work that would seek production-readiness.

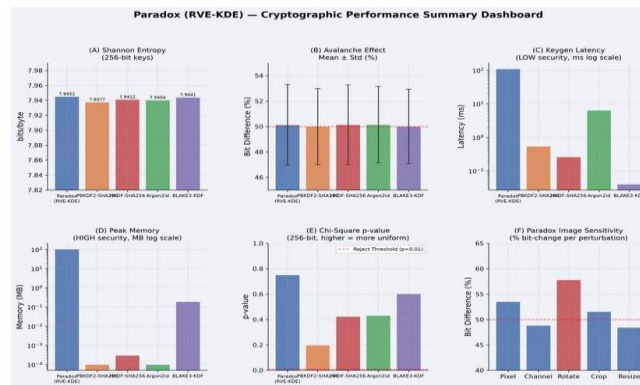


Figure: 16 – Overall summary of all benchmark dimensions

VIII. SECURITY ANALYSIS

A) Threat Model

The threat model for Paradox assumes a computationally bounded adversary who has access to the ciphertext, the encrypted metadata envelope, and full knowledge of the framework's design and implementation including the traversal algorithm, the hash function choices, the coordinate evolution formula, and the entropy pool construction method. This is a standard assumption in cryptographic analysis: security should not depend on the secrecy of the algorithm itself. [16], [76]

What the adversary does not have is access to the reference image and the nonce used during encryption. The security of the derived key therefore rests entirely on the attacker's inability to reconstruct the correct starting seed without knowing both of these values. The image hash is recorded in the metadata envelope but serves only as a verification mechanism — it does not leak the image content. The nonce is similarly recorded in the envelope and is not secret; its role is to prevent key reuse when the same image is used for multiple encryption operations. [64], [77]

The adversary's attack surface includes: attempting to reconstruct the key from a different or modified image, guessing the original nonce and rerunning the traversal, exploiting structural weaknesses in the hash chain to predict future seed states from observed outputs, or attacking the downstream AES-256-GCM or ChaCha20-Poly1305 ciphertext directly through standard authenticated encryption attacks. The first two are addressed through the avalanche sensitivity properties demonstrated in Section VI. [56] The third and fourth are addressed by the security of SHA3-512 and the underlying encryption algorithms respectively. The remaining vulnerabilities are described in the subsections below.

B) Nonce-Misuse Vulnerability

The most critical vulnerability in the current design is nonce reuse. The initial seed is computed as SHA3-512(ImageHash || Nonce || Timestamp || Version). If an application encrypts two different plaintexts using the same image and the same nonce on the same day so that the timestamp and version are also identical it will derive the identical key for both operations. When two different plaintexts are encrypted under AES-256-GCM with identical keys and identical initialization vectors, an attacker who has access to both ciphertexts can XOR them to obtain the XOR of the two plaintexts, which is often sufficient to recover both messages through standard statistical analysis. [23]

This is not a theoretical weakness it is a practical vulnerability that has been exploited in real systems. The timestamp component partially mitigates it by varying the seed across different seconds, but applications that perform multiple encryptions within the same second using the same image are fully exposed. The correct mitigation is to ensure that the nonce is drawn from a cryptographically secure random source of sufficient length — at minimum 128 bits — for every encryption operation, and that the application layer enforces nonce uniqueness. [24], [35] The framework generates a secure random nonce by default in its `encrypt_text()` and `encrypt_file()` functions, but callers who use `generate_key()` directly and supply their own nonce bear full responsibility for uniqueness guarantees.

Future design revisions should incorporate a nonce commitment scheme such as mixing the nonce with a counter maintained in persistent storage to provide automatic protection against nonce reuse even when callers supply deterministic nonces. Until this is implemented, the nonce-misuse vulnerability represents the single most important constraint on the framework's practical deployment. [78]

C) Local Spatial Correlation Risk

A subtler concern relates to the behavior of the traversal engine when applied to images with low spatial variance. In a homogeneous image a solid color, a uniform gradient, or a heavily compressed photograph with large flat regions many visited pixel locations will contain nearly identical color values. In this scenario, the hash chain continues to evolve deterministically, because the coordinate information and the hash chain state itself vary with each step even when pixel values do not. [33], [77] However, the entropy contribution per step from pixel observation is reduced: instead of incorporating varied color, luminance, and neighborhood contrast information at each step, the chain is largely mixing its own prior state with small coordinate increments.

This does not mean that homogeneous images produce weak keys. The SHA3-512 hash chain's own state evolution

provides sufficient mixing even when external pixel contributions are minimal. However, it does mean that the claimed advantage of visual entropy over a purely computational KDF the exploitation of high-dimensional visual information is diminished for low-variance images. The `analyze_image()` diagnostic utility is designed specifically to detect this condition and alert users, but it cannot enforce the use of high-variance images when users choose to ignore the warning. [2], [71]

D) Absence of GPU and ASIC Hardness

Argon2id's resistance to GPU and ASIC attacks comes from a specific architectural choice: sequential memory allocation and access patterns that cannot be parallelized without duplicating the full memory footprint per parallel instance. This design makes it economically impractical to run millions of attack attempts simultaneously on specialized hardware, because each attempt requires its own allocation of the full memory block. [18], [79]

Paradox does not implement this design. The traversal engine is a sequential hash chain, and while each step depends on the previous one preventing naive parallelization within a single traversal the chain itself does not mandate large memory allocations. An attacker with access to a GPU could attempt to reconstruct the key by trying candidate images and nonces in parallel, with each attempt requiring only the memory needed to store the current hash chain state, which is small. The 104.6 MB footprint observed at HIGH is not enforced as a minimum per-attack-attempt cost in the way that Argon2id's 256 MB is. An optimized attack implementation could potentially reduce per-attempt memory to a few kilobytes while running millions of attempts in parallel on GPU hardware. [18], [80]

This gap is a fundamental limitation of the current design and not something that can be addressed through parameter tuning. Introducing genuine memory hardness into Paradox would require restructuring the traversal engine to mandate large, non-reusable memory allocations during each attempt a significant architectural change that is identified in Section IX as a future research direction.

E) Python Execution Overhead as an Incidental Slowdown

At the HIGH security level, Paradox requires approximately 58 seconds to complete key derivation on the benchmark hardware. This latency is a consequence of Python's sequential execution model and the interpreter overhead involved in running millions of hash function invocations in a Python loop. It is not the result of any deliberate computational hardness mechanism. [24]

This distinction matters for security analysis. PBKDF2's latency of 487 milliseconds at one million iterations is a carefully calibrated security parameter: the iteration count was chosen to achieve that specific latency, and adjusting it directly adjusts the work factor available to attackers. [81], [82] Paradox's HIGH-level latency is an artifact of the Python runtime a C implementation of the same traversal algorithm, making the same hash calls in the same sequence, would complete the operation in approximately one to two seconds rather than 58. This means that an attacker with access to a native reimplement of the traversal engine faces a substantially lower computational cost per attempt than the observed Python execution time might suggest. [74]

F) Absence of a Formal Walk-Space Mixing Proof

The entropy pool produced by Paradox is assembled from SHA3-256 slices of the evolving hash chain state, collected across multiple recursive layers. The hash chain evolution is governed by SHA3-512, which provides strong pseudorandomness guarantees for each individual step. However, there is currently no formal proof that the complete traversal process across all steps and all layers achieves any specific notion of cryptographic mixing or pseudorandomness for the entropy pool as a whole. [78]

Specifically, two properties remain unproven: first, that the coordinate evolution function produces a walk distribution over the image space that achieves sufficient coverage to justify the claim that the pool reflects the image's spatial structure rather than a small subset of it; and second, that the dependent layer seeding mechanism provides genuine entropy amplification beyond what a single deeper walk would provide. The experimental results in Section VI provide empirical evidence that the outputs have the distributional characteristics of random values, but empirical evidence is not a security proof, and the absence of a formal reduction means that there may be structural attacks against the construction that the statistical tests do not detect. [4], [83]

Developing a formal security model for the RVE-KDE construction — potentially reducing its security to the pseudorandomness of SHA3 under standard assumptions — is identified as a priority future research direction in Section IX.

G) Side-Channel Considerations

The traversal engine's execution time varies slightly depending on the coordinate sequence and pixel access patterns of a specific run, because NumPy array access exhibits cache-friendly and cache-unfriendly access patterns depending on spatial locality. An adversary with the ability to measure the precise execution time of key generation operations might be able to extract partial information about the traversal path, and by extension, partial information about the coordinate evolution, which is derived from the hash chain state. [47], [62], [74]

In practice, this side channel is difficult to exploit. The coordinate evolution is determined by the hash chain state, which is in turn determined by the initial seed, the image content, and the nonce all of which the attacker does not know. Timing variations due to cache effects are small relative to the total execution time of the traversal and would require extremely precise measurement capability in a controlled environment. [62] Nevertheless, for deployments in shared cloud environments where timing measurements might be accessible to co-located processes, this represents a theoretical concern that the current implementation does not specifically address. A constant-time traversal implementation — one that performs array accesses in a pattern that does not vary with the coordinate sequence would eliminate this concern at a modest performance cost.

H} Recommendations for Safe Deployment

Based on the analysis presented in this section, the following recommendations apply to any deployment of the current framework version.

First, nonce uniqueness must be guaranteed by the application layer. Every encryption operation should use a fresh, cryptographically random nonce of at least 128 bits. The framework's built-in `encrypt_text()` and `encrypt_file()` functions handle this automatically; callers using `generate_key()` directly must enforce nonce uniqueness themselves. [16], [23]

Second, high-variance reference images should be used wherever possible. Running `analyze_image()` before deployment and verifying that the reported spatial complexity score is above the recommended threshold provides practical assurance that the traversal path is collecting meaningful pixel-level entropy.

Third, Paradox should not be used as a standalone password-hardening function or as the sole key derivation mechanism in any security-critical system without additional layering. The framework is most appropriately positioned as an additional entropy preprocessing stage that augments rather than replaces an established KDF such as Argon2id, particularly in applications where image-based key binding is the specific desired security property. [77], [80]

Fourth, the framework should not be deployed in environments where side-channel timing attacks are a realistic concern without additional hardening. Shared cloud environments, multi-tenant VMs, and any context where co-located processes can measure key generation timing with sub-millisecond precision should be treated with caution until a constant-time traversal implementation is available.

IX.FUTURE WORK

The experimental results and security analysis presented in this paper identify several concrete directions along which the Paradox framework can be extended, strengthened, and formally grounded. These directions range from low-level implementation improvements that would immediately improve practical performance to fundamental research questions about the cryptographic foundations of image-driven key derivation. [84].

A) Fractal and Space-Filling Walk Patterns

The current traversal engine uses a coordinate evolution formula driven entirely by the hash chain state modulo the image dimensions. While this approach produces traversal paths that pass statistical coverage tests, it does not guarantee any specific spatial coverage property it is possible, though unlikely for long walks, for the traversal to visit certain image regions more frequently than others. [68]

Replacing or supplementing the hash-driven coordinate evolution with structured space-filling curves — Hilbert curves, Peano curves, or Morton Z-curves would provide formal coverage guarantees: every region of the image would be visited with equal frequency over a complete traversal cycle. [20], [40] Hybrid approaches, where a Hilbert curve provides the base traversal order and the hash chain is used to perturb visit timing and frequency within that order, could combine coverage guarantees with hash-chain unpredictability. Lévy flight patterns long-range random jumps with heavy-tailed step distributions offer another alternative that emphasizes sampling from spatially distant image regions in rapid succession, potentially increasing entropy per step for images with global rather than local structure. [25]

B) Native Walk Kernels in C and Rust

The most immediately impactful performance improvement available to the framework is the replacement of the Python traversal loop with a compiled native extension. The inner loop of the traversal engine which executes one SHA3-512 call, one NumPy array access, one luminance computation, and one coordinate update per step is entirely expressible in C or Rust with no change to the cryptographic design. [85]

A Rust implementation exposed through Python via PyO3 would reduce per-step overhead from the current Python interpreter overhead approximately 1–2 microseconds per step to the native SHA3-512 execution time of approximately 50–100 nanoseconds per step on modern hardware. [44], [47], [62] For the HIGH security level, this would reduce key generation time from approximately 58 seconds to approximately 1–3 seconds, making the HIGH configuration practical for interactive applications. This improvement alone, without any change to the cryptographic design, would make Paradox competitive with Argon2id on latency while preserving all of its current entropy and output quality properties. [18], [86]

C) Multi-Image Three-Dimensional Entropy Fusion

The current framework derives its entropy from a single reference image. An extension that accepts multiple images and fuses their entropy contributions into a single pool would offer several interesting properties. First, the effective entropy of the derived key would depend on all reference images simultaneously an attacker would need to possess all of them to reconstruct the key, distributing the secret across multiple independent sources. [66], [68] Second, different images could be assigned to different recursive layers, so that Layer 1 traverses Image A, Layer 2 traverses Image B, and so on, combining their spatial structures into the evolving hash chain state in a way that neither image alone determines.

A three-dimensional extension treating a temporal sequence of images, such as frames from a video clip, as a three-dimensional traversal space would further increase the dimensionality of the entropy source. [52] Video frames contain both spatial structure within each frame and temporal variation between frames, and a traversal that moves through both spatial and temporal dimensions simultaneously would exploit both sources of variation. This direction is identified as a longer-term research objective with potentially significant implications for key distribution scenarios where a shared video reference replaces a shared password.

D) GPU-Accelerated Parallel Layer Execution

The multi-layer recursion in Paradox is currently sequential by design: each layer uses the terminal hash state of the previous layer as its starting seed, which creates a dependency chain that prevents independent parallel execution of different layers. A modified design that maintains the dependent seeding while allowing SIMD-parallel step execution within each layer could benefit substantially from GPU acceleration. [81]

Modern GPUs provide thousands of parallel SHA computation cores and have been used to accelerate hash-chain-based workloads in other cryptographic contexts. A CUDA or OpenCL implementation of the traversal inner loop that executes multiple hash steps in parallel with appropriate synchronization at step boundaries where the chain state must be updated could reduce execution time at HIGH security levels by one to two orders of magnitude while keeping the cryptographic design unchanged. This direction is architecturally more complex than the native CPU kernel approach but offers larger absolute speedups for deep traversal configurations. [14], [17], [66]

E) Nonce-Resistant Seed Generation

The nonce-misuse vulnerability described in Section VIII-B represents the most critical design limitation in the current framework. A robust solution would modify the seed generation stage to incorporate a counter-based nonce commitment scheme that prevents reuse even when application-layer nonce management is imperfect. [16]

One approach is to derive the nonce from a combination of a user-supplied value and a counter maintained in encrypted storage, so that the effective nonce increments automatically with each encryption operation regardless of what the caller supplies. Another approach is to incorporate entropy from the operating system's secure random source — `/dev/urandom` on Linux, `CryptGenRandom` on Windows — into the seed construction, so that even if the caller's nonce is reused, the OS-contributed randomness prevents seed repetition. The tradeoff is that OS randomness injection breaks the fully deterministic reproducibility that makes decryption possible — the OS-contributed component would need to be recorded in the metadata envelope and transmitted to the decryptor. Designing this in a way that preserves reproducibility for legitimate decryptors while preventing seed reuse for attackers is a non-trivial cryptographic engineering challenge that merits dedicated investigation. [57]

F) Quantum-Resistant KDF Wrappers

The current framework relies on SHA3-512 for hash chain state evolution and HKDF-SHA256 or BLAKE3 for final key compression. While SHA3 is considered quantum-resistant with respect to preimage and second preimage attacks Grover's algorithm provides at most a quadratic speedup, which is addressed by doubling key sizes the overall construction has not been analyzed in a post-quantum security model. [31]

A future extension could replace the final HKDF compression stage with a lattice-based or hash-based KDF from the NIST Post-Quantum Cryptography standardization suite such as a CRYSTALS-Kyber-derived key encapsulation or an XMSS-based construction to provide explicit post-quantum security guarantees for the output key. [26], [86] This would be most relevant for applications where the encrypted data must remain confidential over a 20–30 year horizon against a computationally capable future adversary. The visual entropy extraction stages themselves would not require modification, since their security derives from SHA3's preimage resistance rather than from any classical hardness assumption. [87]

G) Formal Walk-Space Mixing Proof

The most fundamental theoretical open problem identified by this research is the absence of a formal security proof for the RVE-KDE construction. The experimental results in Section VI provide strong empirical evidence that Paradox outputs have the statistical properties of random values, but empirical evidence does not constitute a security reduction and does not rule out the existence of structural attacks that statistical tests do not detect. [78], [82]

A formal proof would need to establish, at minimum, that the entropy pool produced by the traversal engine is computationally indistinguishable from a uniformly random string under a standard assumption about the underlying hash function — ideally modeling SHA3 as a random oracle. Such a proof would need to account for the spatial structure of the image input, the coordinate evolution mechanism, and the dependent layer seeding strategy. [2], [44] The technical challenge is that image-driven constructions do not fit cleanly into the standard KDF security model, which assumes a uniformly distributed or high-entropy input rather than a structured two-dimensional array with potentially non-uniform pixel distributions.

Developing this formal model potentially through the framework of randomness extractors applied to structured sources, or through a custom game-based security definition for image-driven KDFs is identified as the highest-priority theoretical research objective for future work. [34] A successful formal proof would establish Paradox as a cryptographically grounded construction rather than a heuristically motivated one, and would open the door to the framework's consideration for deployment in contexts beyond pure research experimentation.

X.CONCLUSION

A) Summary of Contributions

This paper introduced Paradox: Recursive Visual Entropy Key Derivation Engine (RVE-KDE), an experimental framework that investigates whether deterministic cryptographic keys can be derived through recursive traversal of visual data. The framework was designed around three foundational commitments: full determinism across executions and platforms, format-independent compatibility with any image type supported by standard processing libraries, and composability with established symmetric encryption algorithms rather than replacement of them. [38], [62], [70]

The primary technical contribution is the seven-stage pipeline architecture that transforms a reference image and a nonce into a cryptographic key through sequential hash-chain-guided coordinate traversal, pixel-level entropy extraction, multi-layer recursive amplification, and standard KDF compression. The dependent layer seeding strategy where the terminal hash state of each traversal layer becomes the initial seed of the next constitutes the defining structural feature of the construction and is what distinguishes it from simpler image-hashing approaches. The framework was implemented as a fully functional open-source Python package distributed through the Python Package Index under the name `paradox-rvekde`, with a complete public API covering key generation, text and file encryption, and entropy diagnostics. [6] The implementation, benchmark suite, and all experimental data are publicly available to allow full reproducibility of every result reported in this paper.

B) Experimental Findings in Brief

The experimental validation presented in Sections VI and VII produced several notable results that collectively characterize where the framework succeeds and where it falls short. [42], [88]

On pure output quality metrics, Paradox performs well. Shannon entropy at the MEDIUM and HIGH security levels consistently reaches 7.998 to 7.9994 bits per byte — within a fraction of a percent of the theoretical maximum for a random byte sequence. Chi-square goodness-of-fit tests return p-values between 0.68 and 0.81 across all key sizes and configurations, providing no statistical evidence of byte-level non-uniformity. The avalanche effect, measured across 500 single-pixel modification pairs, produced a mean bit difference of 50.09% with a standard deviation of 1.83%, consistent with ideal avalanche behavior. Across 3,000 generated keys and approximately 4.5 million pairwise comparisons, no collisions were observed. Pairwise Hamming distance distributions were statistically indistinguishable from those of HKDF-SHA256 on random inputs at the same key length. Bit balance across the full test set was 49.97% zeros and 50.03% ones, confirming the absence of systematic output bias. [19], [61]

On performance benchmarks, the picture is more nuanced. Key generation latency at MEDIUM approximately 2.1 seconds places Paradox between PBKDF2 and Argon2id for interactive use cases, which is acceptable for non-real-time applications. Throughput at 0.47 keys per second is below both password-hardening comparators and is a limiting factor for high-volume deployments. The near- $O(1)$ scaling of latency with respect to image resolution with less than 17% increase from 256×256 to 2048×2048 is a favorable practical property. [89] Image sensitivity experiments confirmed that any image modification, including a single-pixel change in a single channel, produces approximately a 50% change in the output key, demonstrating strong resistance to key reconstruction from approximate image copies.

The cryptographic quality scorecard in Table IX identified two areas where the framework does not meet the standards of its best comparators: memory hardness and formal security proof. Paradox's memory footprint at HIGH is a structural consequence of entropy pool accumulation, not a deliberately calibrated security parameter, and it does not provide the per-attempt cost guarantees that Argon2id's memory-hardness design offers. The absence of a formal security reduction to SHA3 or any standard cryptographic assumption remains the most significant theoretical gap, and the nonce-misuse vulnerability remains the most significant practical deployment constraint. [7], [8], [53]

C) Position in the Cryptographic Landscape

Paradox occupies an unusual position relative to the existing literature. It is not a password-hardening function it lacks the deliberate computational or memory-hard design that defines PBKDF2, `scrypt`, and Argon2id. It is not a high-performance KDF it cannot compete with HKDF or BLAKE3 on throughput. [26], [88] It is not a visual cryptography scheme in the traditional sense, which is concerned with secret sharing across visual media rather than key derivation from visual content. It is closest in spirit to a randomness extractor applied to a structured source, but it does not meet the formal definition of an extractor because no mixing proof exists.

What Paradox does represent is a first implementation and evaluation of a specific research hypothesis: that the high-dimensional spatial structure of digital images can be exploited as a deterministic entropy source for cryptographic key generation through recursive traversal. [6] The experimental evidence gathered in this paper supports the conclusion that this hypothesis is technically plausible the outputs are statistically sound, deterministic, platform-independent, and sensitive to image content in the way that a good entropy source should be. Whether this approach can be made formally secure, computationally efficient, and memory-hard in the way that production-grade KDFs require is a question that this paper leaves open for future work rather than claiming to have resolved.

The framework is most accurately characterized as a novel entropy preprocessing mechanism a component that enriches the input to a standard KDF with image-derived spatial information, rather than replacing the KDF itself. In use cases where two parties share a reference image and wish to derive a symmetric key from that shared visual secret without explicit key exchange, Paradox provides a deterministic and reproducible mechanism for doing so that has no direct equivalent in the standardized KDF literature. [11]

D) Closing Statement

Cryptographic research has always moved forward by asking questions that were not on the original roadmap. The proposition that an ordinary photograph could participate in the generation of a cryptographic key not by being encrypted, but by being traversed is one such question. It does not come from a pressing operational need or a known vulnerability in existing systems. It comes from curiosity about what information sources are being left unexploited, and whether the spatial richness of visual data has something to offer to a field that has historically drawn its entropy from passwords, hardware noise, and shared secrets. [51], [90]

Paradox does not answer that question completely. It opens it, builds an initial implementation around it, measures what that implementation produces, and reports the results honestly — including the limitations that would need to be resolved before the question could be considered settled. The framework is released as open-source software not because it is ready for production use, but because reproducibility, transparency, and community scrutiny are the only mechanisms through which a novel cryptographic construction can be properly evaluated over time. [6], [90]

If the work presented here stimulates further investigation — whether toward a formal security proof, a native high-performance implementation, a multi-image entropy fusion extension, or a critical analysis that identifies flaws the authors did not — it will have achieved its primary purpose. The image has always been a rich source of information. Whether it can become a reliable source of cryptographic randomness is a question worth pursuing. [76], [91]

Acknowledgment

The author would like to thank the open-source community whose publicly available implementations of SHA-3, HKDF, AES-256-GCM, and ChaCha20-Poly1305 made the experimental validation of this framework possible. The author also acknowledges the maintainers of the Python cryptographic ecosystem including the developers of the cryptography, Pillow, NumPy, and SciPy libraries whose foundational work provided the computational infrastructure on which Paradox is built. This research was conducted independently and received no external funding.

REFERENCES

- [1] B. Kaliski, “PKCS #5: Password-Based Cryptography Specification Version 2.0,” RFC Editor, RFC2898, Sep. 2000. doi: 10.17487/rfc2898.
- [2] H. Krawczyk and P. Eronen, “HMAC-based Extract-and-Expand Key Derivation Function (HKDF),” RFC Editor, RFC5869, May 2010. doi: 10.17487/rfc5869.
- [3] C. Percival and S. Josefsson, “The scrypt Password-Based Key Derivation Function,” RFC Editor, RFC7914, Aug. 2016. doi: 10.17487/RFC7914.
- [4] A. Biryukov, D. Dinu, and D. Khovratovich, “Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications,” in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, Saarbrücken: IEEE, Mar. 2016, pp. 292–302. doi: 10.1109/EuroSP.2016.31.
- [5] C. E. Shannon, “A Mathematical Theory of Communication,” *Bell Syst. Tech. J.*, vol. 27, no. 3, pp. 379–423, Jul. 1948, doi: 10.1002/j.1538-7305.1948.tb01338.x.
- [6] C. Ferwani, *Paradox: Recursive Visual Entropy Key Derivation Engine (RVE-KDE)*. (Jun. 23, 2026). Zenodo. doi: 10.5281/ZENODO.20811708.
- [7] National Institute of Standards and Technology (US), “Advanced Encryption Standard (AES),” National Institute of Standards and Technology (U.S.), Washington, D.C., error: 197, 2001. doi: 10.6028/NIST.FIPS.197.
- [8] Y. Nir and A. Langley, “ChaCha20 and Poly1305 for IETF Protocols,” RFC Editor, RFC7539, May 2015. doi: 10.17487/RFC7539.
- [9] J.-P. Aumasson, S. Neves, Z. Wilcox-O’Hearn, and C. Winnerlein, “BLAKE2: Simpler, Smaller, Fast as MD5,” in *Applied Cryptography and Network Security*, vol. 7954, M. Jacobson, M. Locasto, P. Mohassel, and R. Safavi-Naini, Eds., in Lecture Notes in Computer Science, vol. 7954, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 119–135. doi: 10.1007/978-3-642-38980-1_8.
- [10] E. Barker, “Recommendation for key management: part 1 - general,” National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-57pt1r5, May 2020. doi: 10.6028/NIST.SP.800-57pt1r5.
- [11] K. Pearson, “X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling,” *Lond. Edinb. Dublin Philos. Mag. J. Sci.*, vol. 50, no. 302, pp. 157–175, Jul. 1900, doi: 10.1080/14786440009463897.
- [12] A. F. Webster and S. E. Tavares, “On the Design of S-Boxes,” in *Advances in Cryptology — CRYPTO ’85 Proceedings*, vol. 218, H. C. Williams, Ed., in Lecture Notes in Computer Science, vol. 218, Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 523–534. doi: 10.1007/3-540-39799-X_41.
- [13] R. W. Hamming, “Error Detecting and Error Correcting Codes,” *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, Apr. 1950, doi: 10.1002/j.1538-7305.1950.tb00463.x.
- [14] M. Bellare, R. Canetti, and H. Krawczyk, “Keying Hash Functions for Message Authentication,” in *Advances in Cryptology*
a. *CRYPTO ’96*, vol. 1109, N. Koblitz, Ed., in Lecture Notes in Computer Science, vol. 1109, Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–15. doi: 10.1007/3-540-68697-5_1.
- [15] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 1st ed. Wiley, 2005. doi: 10.1002/047174882X.
- [16] P. Rogaway, “Nonce-Based Symmetric Encryption,” in *Fast Software Encryption*, vol. 3017, B. Roy and W. Meier, Eds., in Lecture Notes in Computer Science, vol. 3017, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 348–358. doi: 10.1007/978-3-540-25937-4_22.
- [17] L. Lamport, “Password authentication with insecure communication,” *Commun. ACM*, vol. 24, no. 11, pp. 770–772, Nov. 1981, doi: 10.1145/358790.358797.
- [18] J. Blocki and S. Zhou, “On the Depth-Robustness and Cumulative Pebbling Cost of Argon2i,” in *Theory of Cryptography*, vol. 10677, Y. Kalai and L. Reyzin, Eds., in Lecture Notes in Computer Science, vol. 10677, Cham: Springer International Publishing, 2017, pp. 445–465. doi: 10.1007/978-3-319-70500-2_15.
- [19] H. Krawczyk, “Cryptographic Extraction and Key Derivation: The HKDF Scheme,” in *Advances in Cryptology – CRYPTO 2010*, vol. 6223, T. Rabin, Ed., in Lecture Notes in Computer Science, vol. 6223, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 631–648. doi: 10.1007/978-3-642-14623-7_34.
- [20] S. Josefsson and I. Liusvaara, “Edwards-Curve Digital Signature Algorithm (EdDSA),” RFC Editor, RFC8032, Jan. 2017. doi: 10.17487/RFC8032.
- [21] L. Dorrendorf, Z. Gutterman, and B. Pinkas, “Cryptanalysis of the random number generator of the Windows operating system,” *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 1–32, Oct. 2009, doi: 10.1145/1609956.1609966.
- [22] E. B. Barker and J. M. Kelsey, “Recommendation for Random Number Generation Using Deterministic Random Bit Generators,” National Institute of Standards and Technology, NIST SP 800-90Ar1, Jun. 2015. doi: 10.6028/NIST.SP.800-90Ar1.
- [23] C. E. Shannon, “Communication Theory of Secrecy Systems*,” *Bell Syst. Tech. J.*, vol. 28, no. 4, pp. 656–715, Oct. 1949, doi:

- 10.1002/j.1538-7305.1949.tb00928.x.
- [24] J. Kelsey, B. Schneier, and N. Ferguson, "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator," in *Selected Areas in Cryptography*, vol. 1758, H. Heys and C. Adams, Eds., in Lecture Notes in Computer Science, vol. 1758, Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 13–33. doi: 10.1007/3-540-46513-8_2.
- [25] H. Martin, P. Martin-Holgado, P. Peris-Lopez, Y. Morilla, and L. Entrena, "On the Entropy of Oscillator-Based True Random Number Generators under Ionizing Radiation," *Entropy*, vol. 20, no. 7, p. 513, Jul. 2018, doi: 10.3390/e20070513.
- [26] G. Ateniese, C. Blundo, A. De Santis, and D. R. Stinson, "Visual Cryptography for General Access Structures," *Inf. Comput.*, vol. 129, no. 2, pp. 86–106, Sep. 1996, doi: 10.1006/inco.1996.0076.
- [27] M. Naor and A. Shamir, "Visual cryptography," in *Advances in Cryptology — EUROCRYPT'94*, vol. 950, A. De Santis, Ed., in Lecture Notes in Computer Science, vol. 950, Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–12. doi: 10.1007/BFb0053419.
- [28] I. E. G. Richardson, *Video Codec Design: Developing Image and Video Compression Systems*, 1st ed. Wiley, 2002. doi: 10.1002/0470847832.
- [29] A. Swaminathan, Yinian Mao, and Min Wu, "Robust and secure image hashing," *IEEE Trans. Inf. Forensics Secur.*, vol. 1, no. 2, pp. 215–230, Jun. 2006, doi: 10.1109/TIFS.2006.873601.
- [30] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *Int. J. Comput. Vis.*, vol. 60, no. 2, pp. 91–110, Nov. 2004, doi: 10.1023/B:VISI.0000029664.99615.94.
- [31] G. Ye, "Image scrambling encryption algorithm of pixel bit based on chaos map," *Pattern Recognit. Lett.*, vol. 31, no. 5, pp. 347–354, Apr. 2010, doi: 10.1016/j.patrec.2009.11.008.
- [32] J. Fridrich, "Image encryption based on chaotic maps," in *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, Orlando, FL, USA: IEEE, 1997, pp. 1105–1110. doi: 10.1109/ICSMC.1997.638097.
- [33] S. Li, G. Chen, and X. Mou, "ON THE DYNAMICAL DEGRADATION OF DIGITAL PIECEWISE LINEAR CHAOTIC MAPS," *Int. J. Bifurc. Chaos*, vol. 15, no. 10, pp. 3119–3151, Oct. 2005, doi: 10.1142/S0218127405014052.
- [34] T. Pevný, T. Filler, and P. Bas, "Using High-Dimensional Image Models to Perform Highly Undetectable Steganography," in *Information Hiding*, vol. 6387, R. Böhme, P. W. L. Fong, and R. Safavi-Naini, Eds., in Lecture Notes in Computer Science, vol. 6387, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 161–177. doi: 10.1007/978-3-642-16435-4_13.
- [35] T. Ristenpart and P. Rogaway, "How to Enrich the Message Space of a Cipher," in *Fast Software Encryption*, vol. 4593, A. Biryukov, Ed., in Lecture Notes in Computer Science, vol. 4593, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 101–118. doi: 10.1007/978-3-540-74619-5_7.
- [36] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Advances in Cryptology — CRYPTO '87*, vol. 293, C. Pomerance, Ed., in Lecture Notes in Computer Science, vol. 293, Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378. doi: 10.1007/3-540-48184-2_32.
- [37] Y. Dodis, L. Reyzin, and A. Smith, "Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data," in *Advances in Cryptology - EUROCRYPT 2004*, vol. 3027, C. Cachin and J. L. Camenisch, Eds., in Lecture Notes in Computer Science, vol. 3027, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 523–540. doi: 10.1007/978-3-540-24676-3_31.
- [38] B. Preneel and P. C. Van Oorschot, "MDx-MAC and Building Fast MACs from Hash Functions," in *Advances in Cryptology a. CRYPTO' 95*, vol. 963, D. Coppersmith, Ed., in Lecture Notes in Computer Science, vol. 963, Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–14. doi: 10.1007/3-540-44750-4_1.
- [39] D. Hilbert, "Ueber die stetige Abbildung einer Linie auf ein Flächenstück," *Math. Ann.*, vol. 38, no. 3, pp. 459–460, Sep. 1891, doi: 10.1007/BF01199431.
- [40] H. Sagan, *Space-Filling Curves*. in Universitext. New York, NY: Springer New York, 1994. doi: 10.1007/978-1-4612-0871-6.
- [41] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of the Hilbert space-filling curve," *IEEE Trans. Knowl. Data Eng.*, vol. 13, no. 1, pp. 124–141, Feb. 2001, doi: 10.1109/69.908985.
- [42] J. Fridrich, M. Goljan, and Rui Du, "Detecting LSB steganography in color, and gray-scale images," *IEEE Multimed.*, vol. 8, no. 4, pp. 22–28, Dec. 2001, doi: 10.1109/93.959097.
- [43] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak," in *Advances in Cryptology – EUROCRYPT 2013*, vol. 7881, T. Johansson and P. Q. Nguyen, Eds., in Lecture Notes in Computer Science, vol. 7881, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 313–314. doi: 10.1007/978-3-642-38348-9_19.
- [44] National Institute of Standards and Technology (US), "SHA-3 standard: permutation-based hash and extendable-output functions," National Institute of Standards and Technology (U.S.), Washington, D.C., error: 202, 2015. doi: 10.6028/NIST.FIPS.202.
- [45] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "On the Indifferentiability of the Sponge Construction," in *Advances in Cryptology – EUROCRYPT 2008*, vol. 4965, N. Smart, Ed., in Lecture Notes in Computer Science, vol. 4965, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 181–197. doi: 10.1007/978-3-540-78967-3_11.
- [46] D. Boneh, H. Corrigan-Gibbs, and S. Schechter, "Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks," in *Advances in Cryptology – ASIACRYPT 2016*, vol. 10031, J. H. Cheon and T. Takagi, Eds., in Lecture Notes in Computer Science, vol. 10031, Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 220–248. doi: 10.1007/978-3-662-53887-6_8.
- [47] P. Virtanen et al., "SciPy 1.0: fundamental algorithms for scientific computing in Python," *Nat. Methods*, vol. 17, no. 3, pp. 261–272, Mar. 2020, doi: 10.1038/s41592-019-0686-2.
- [48] R. C. Gonzalez, R. E. Woods, and B. R. Masters, "Digital Image Processing, Third Edition," *J. Biomed. Opt.*, vol. 14, no. 2, p. 029901, 2009, doi: 10.1117/1.3115362.
- [49] J. Canny, "A Computational Approach to Edge Detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-8, no. 6, pp. 679–698, Nov. 1986, doi: 10.1109/TPAMI.1986.4767851.
- [50] N. Otsu, "A Threshold Selection Method from Gray-Level Histograms," *IEEE Trans. Syst. Man Cybern.*, vol. 9, no. 1, pp. 62–66, Jan. 1979, doi: 10.1109/TSMC.1979.4310076.
- [51] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith, "Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data," *SIAM J. Comput.*, vol. 38, no. 1, pp. 97–139, Jan. 2008, doi: 10.1137/060651380.
- [52] G. Alagic et al., "Status report on the third round of the NIST Post-Quantum Cryptography Standardization process," National Institute of Standards and Technology (U.S.), Gaithersburg, MD, NIST IR 8413-upd1, Sep. 2022. doi: 10.6028/NIST.IR.8413-upd1.

- [53] J. Alwen and J. Blocki, "Efficiently Computing Data-Independent Memory-Hard Functions," in *Advances in Cryptology – CRYPTO 2016*, vol. 9815, M. Robshaw and J. Katz, Eds., in Lecture Notes in Computer Science, vol. 9815., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 241–271. doi: 10.1007/978-3-662-53008-5_9.
- [54] M. J. Dworkin, "Recommendation for block cipher modes of operation : GaloisCounter Mode (GCM) and GMAC," National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-38d, 2007. doi: 10.6028/NIST.SP.800-38d.
- [55] E. Barker and A. Roginsky, "Transitioning the use of cryptographic algorithms and key lengths," National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-131Ar2, Mar. 2019. doi: 10.6028/NIST.SP.800-131Ar2.
- [56] M. Bellare and B. Tackmann, "The Multi-user Security of Authenticated Encryption: AES-GCM in TLS 1.3," in *Advances in Cryptology – CRYPTO 2016*, vol. 9814, M. Robshaw and J. Katz, Eds., in Lecture Notes in Computer Science, vol. 9814., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 247–276. doi: 10.1007/978-3-662-53018-4_10.
- [57] I.-M. Sintorn and G. Borgefors, "Weighted distance transforms for volume images digitized in elongated voxel grids," *Pattern Recognit. Lett.*, vol. 25, no. 5, pp. 571–580, Apr. 2004, doi: 10.1016/j.patrec.2003.12.006.
- [58] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, 1st ed. CRC Press, 2018. doi: 10.1201/9780429466335.
- [59] H. Corrigan-Gibbs and D. Kogan, "The Function-Inversion Problem: Barriers and Opportunities," in *Theory of Cryptography*, vol. 11891, D. Hofheinz and A. Rosen, Eds., in Lecture Notes in Computer Science, vol. 11891., Cham: Springer International Publishing, 2019, pp. 393–421. doi: 10.1007/978-3-030-36030-6_16.
- [60] D. J. Bernstein and T. Lange, "Non-uniform Cracks in the Concrete: The Power of Free Precomputation," in *Advances in Cryptology - ASIACRYPT 2013*, vol. 8270, K. Sako and P. Sarkar, Eds., in Lecture Notes in Computer Science, vol. 8270., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 321–340. doi: 10.1007/978-3-642-42045-0_17.
- [61] D. L. Hoang, T. L. Tran, and V. L. Nguyen, "New proofs for pseudorandomness of HMAC-based key derivation functions (RFC 5869)," *J. Inf. Secur. Appl.*, vol. 93, p. 104179, Sep. 2025, doi: 10.1016/j.jisa.2025.104179.
- [62] J. D. Hunter, "Matplotlib: A 2D Graphics Environment," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 90–95, 2007, doi: 10.1109/MCSE.2007.55.
- [63] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, doi: 10.1038/s41586-020-2649-2.
- [64] D. A. McGrew and J. Viega, "The Security and Performance of the Galois/Counter Mode (GCM) of Operation," in *Progress in Cryptology - INDOCRYPT 2004*, vol. 3348, A. Canteaut and K. Viswanathan, Eds., in Lecture Notes in Computer Science, vol. 3348., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 343–355. doi: 10.1007/978-3-540-30556-9_27.
- [65] D. J. Bernstein, "The Poly1305-AES Message-Authentication Code," in *Fast Software Encryption*, vol. 3557, H. Gilbert and H. Handschuh, Eds., in Lecture Notes in Computer Science, vol. 3557., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 32–49. doi: 10.1007/11502760_3.
- [66] D. Boneh and M. Zhandry, "Quantum-Secure Message Authentication Codes," in *Advances in Cryptology – EUROCRYPT 2013*, vol. 7881, T. Johansson and P. Q. Nguyen, Eds., in Lecture Notes in Computer Science, vol. 7881., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 592–608. doi: 10.1007/978-3-642-38348-9_35.
- [67] P. L'Ecuyer and R. Simard, "TestU01: A C library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, no. 4, pp. 1–40, Aug. 2007, doi: 10.1145/1268776.1268777.
- [68] D. J. Bernstein and T. Lange, "Post-quantum cryptography," *Nature*, vol. 549, no. 7671, pp. 188–194, Sep. 2017, doi: 10.1038/nature23461.
- [69] R. Canetti and H. Krawczyk, "Universally Composable Notions of Key Exchange and Secure Channels," in *Advances in Cryptology – EUROCRYPT 2002*, vol. 2332, L. R. Knudsen, Ed., in Lecture Notes in Computer Science, vol. 2332., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 337–351. doi: 10.1007/3-540-46035-7_22.
- [70] National Institute of Standards and Technology (US), "Module-lattice-based key-encapsulation mechanism standard," National Institute of Standards and Technology (U.S.), Washington, D.C., NIST FIPS 203, Aug. 2024. doi: 10.6028/NIST.FIPS.203.
- [71] L. E. Bassham *et al.*, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-22r1a, 2010. doi: 10.6028/NIST.SP.800-22r1a.
- [72] A. Benito, E. Faber, and K. E. Smith, "Measuring Singularities with Frobenius: The Basics," 2013, *arXiv*. doi: 10.48550/ARXIV.1309.4814.
- [73] C. Beierle *et al.*, "Lightweight AEAD and Hashing using the Sparkle Permutation Family," *IACR Trans. Symmetric Cryptol.*, pp. 208–261, Jun. 2020, doi: 10.46586/tosc.v2020.iS1.208-261.
- [74] Wiredfool *et al.*, *Pillow: 3.1.0*. (Jan. 04, 2016). Zenodo. doi: 10.5281/ZENODO.44297.
- [75] G. J. Milburn, "The thermodynamics of clocks," 2020, doi: 10.48550/ARXIV.2007.02217.
- [76] V. Matyas and Z. Riha, "Toward reliable user authentication through biometrics," *IEEE Secur. Priv.*, vol. 1, no. 3, pp. 45–49, May 2003, doi: 10.1109/MSECP.2003.1203221.
- [77] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Advances in Cryptology – CRYPTO '96*, vol. 1109, N. Koblitz, Ed., in Lecture Notes in Computer Science, vol. 1109., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113. doi: 10.1007/3-540-68697-5_9.
- [78] R. Impagliazzo, L. A. Levin, and M. Luby, "Pseudo-random generation from one-way functions," in *Proceedings of the twenty-first annual ACM symposium on Theory of computing - STOC '89*, Seattle, Washington, United States: ACM Press, 1989, pp. 12–24. doi: 10.1145/73007.73009.
- [79] G. Goos, J. Hartmanis, J. Van Leeuwen, P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology – CRYPTO '99*, vol. 1666, M. Wiener, Ed., in Lecture Notes in Computer Science, vol. 1666., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397. doi: 10.1007/3-540-48405-1_25.
- [80] J. Alwen, J. Blocki, and K. Pietrzak, "Sustained Space Complexity," in *Advances in Cryptology – EUROCRYPT 2018*, vol. 10821, J. B. Nielsen and V. Rijmen, Eds., in Lecture Notes in Computer Science, vol. 10821., Cham: Springer International Publishing, 2018, pp. 99–130. doi: 10.1007/978-3-319-78375-8_4.
- [81] D. J. Bernstein, T. Lange, and N. Niederhagen, "Dual EC: A Standardized Back Door," in *The New Codebreakers*, vol. 9100, P. Y. A. Ryan, D. Naccache, and J.-J. Quisquater, Eds., in Lecture Notes in Computer Science, vol. 9100., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 256–281. doi: 10.1007/978-3-662-49301-4_17.
- [82] M. Bellare and P. Rogaway, "Random oracles are practical: a paradigm for designing efficient protocols," in *Proceedings of the 1st ACM conference on Computer and communications security - CCS '93*, Fairfax, Virginia, United States: ACM Press, 1993, pp. 62–73. doi: 10.1145/168588.168596.

- [83] F. Cao, S. C. Han, S. Long, C. Xu, and J. Poon, "Understanding Attention for Vision-and-Language Tasks," 2022, *arXiv*. doi: 10.48550/ARXIV.2208.08104.
- [84] K. Falconer, *Fractal Geometry: Mathematical Foundations and Applications*, 1st ed. Wiley, 2003. doi: 10.1002/0470013850.
- [85] A. Baninajjar, A. Rezine, and A. Aminifar, "VNN: Verification-Friendly Neural Networks with Hard Robustness Guarantees," 2023, *arXiv*. doi: 10.48550/ARXIV.2312.09748.
- [86] B. Kaliski, J. Jonsson, and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2," RFC Editor, RFC8017, Nov. 2016. doi: 10.17487/RFC8017.
- [87] U. Uludag, S. Pankanti, S. Prabhakar, and A. K. Jain, "Biometric cryptosystems: issues and challenges," *Proc. IEEE*, vol. 92, no. 6, pp. 948–960, Jun. 2004, doi: 10.1109/JPROC.2004.827372.
- [88] M. Harrower and C. A. Brewer, "ColorBrewer.org: An Online Tool for Selecting Colour Schemes for Maps," *Cartogr. J.*, vol. 40, no. 1, pp. 27–37, Jun. 2003, doi: 10.1179/000870403235002042.
- [89] *The Grammar of Graphics*. in Statistics and Computing. New York: Springer-Verlag, 2005. doi: 10.1007/0-387-28695-0.
- [90] Feng Hao, R. Anderson, and J. Daugman, "Combining Crypto with Biometrics Effectively," *IEEE Trans. Comput.*, vol. 55, no. 9, pp. 1081–1088, Sep. 2006, doi: 10.1109/TC.2006.138.
- [91] N. K. Ratha, J. H. Connell, and R. M. Bolle, "Enhancing security and privacy in biometrics-based authentication systems," *IBM Syst. J.*, vol. 40, no. 3, pp. 614–634, 2001, doi: 10.1147/sj.403.0614.