**A Critical Review of Client-Side Cryptographic Overlays for Enhancing Data Privacy in Public Cloud Storage**

**The Trust Deficit in Public Cloud Storage: A Security Analysis**

The widespread adoption of public cloud storage services, such as Google Drive and Dropbox, has revolutionized data management for individuals and enterprises, offering unparalleled convenience, scalability, and accessibility.1 However, this paradigm shift introduces a fundamental security challenge rooted in the provider-centric trust model. This section deconstructs the security architecture of major cloud providers to establish the foundational problem this dissertation addresses: the inherent "trust issue" that arises when users relinquish direct control over their data's confidentiality.1

**The Provider-Centric Security Model**

Major public cloud storage providers (CSPs) like Google and Dropbox have invested heavily in securing their global infrastructure. Their security models are predicated on robust, multi-layered defences designed to protect user data from external threats.2 Core to this model are the practices of encryption at rest and encryption in transit. Data stored on provider servers (at rest) is typically encrypted using strong symmetric algorithms like 256-bit Advanced Encryption Standard (AES).5 Similarly, data moving between the user's client and the provider's servers (in transit) is protected using protocols like Transport Layer Security (TLS).4

While these measures are crucial for protecting against infrastructure-level attacks, they share a critical characteristic: the cloud service provider manages the encryption keys.8 Google, for instance, explicitly states that for its standard services, it holds the encryption keys, which means it technically has the ability to access user files.10 Dropbox operates on a similar model, managing encryption keys on behalf of users to enable product features.4 This architecture positions the CSP as a data custodian rather than a zero-knowledge service provider. The user must trust that the provider will not access their data, that its internal controls are impervious to malicious insiders, and that its systems cannot be compromised in a way that exposes plaintext data.

The existence of premium, enterprise-focused "Client-Side Encryption" (CSE) offerings from these same providers serves as a tacit acknowledgment of this trust deficit. Google Workspace, for example, offers an optional CSE feature where the customer holds the encryption keys, explicitly providing an "additional layer of security and privacy for sensitive data".6 Similarly, Dropbox's strategic acquisition of Boxcryptor, a leading CSE provider, was aimed at integrating end-to-end encryption capabilities for its business users.8 This market trend directly validates the problem statement: the standard, widely-used cloud storage model is recognized, even by the providers themselves, as insufficient for users with high confidentiality requirements. This creates a significant gap for individuals and Small to Medium-sized Enterprises (SMEs) who require true privacy but may not have access to or the budget for these enterprise-grade features.1

**Threat Vectors in a Trusted Third-Party Model**

The provider-centric security model, which relies on a trusted third party, is vulnerable to several threat vectors that client-side encryption is designed to mitigate. A formal threat model, such as STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege), can be applied to analyze these risks.13

* **Information Disclosure**: This is the most significant threat. Because the provider holds the keys, several actors could potentially gain access to plaintext data.
  + **Malicious Insider**: A disgruntled or coerced employee with sufficient privileges could access sensitive user data.1 While providers implement strict internal controls and background checks 3, the risk is inherent to the architectural model.
  + **Compromised Provider Systems**: A sophisticated external attacker who successfully breaches the CSP's infrastructure could potentially gain access to the key management systems and, consequently, decrypt user data.1 Historical data breaches, including a significant incident at Dropbox where user credentials were leaked, demonstrate that even major providers are not immune to such attacks.8
  + **Governmental and Legal Compulsion**: Legal frameworks can compel a CSP to disclose user data to government or law enforcement agencies, often without the user's knowledge or consent.18 If the provider holds the decryption keys, it can be forced to provide data in its unencrypted form.
* **Tampering**: Although CSPs implement internal mechanisms to ensure data integrity, these are not independently verifiable by the user. A highly sophisticated internal attacker or a compromised system could, in theory, subtly alter user data without detection.1 The user lacks a cryptographic guarantee, under their own control, that the data they download is identical to the data they uploaded.
* **Spoofing**: An attacker who compromises the provider's central authentication system could potentially impersonate a legitimate user and gain full access to their unencrypted data stored on the service.

Academic research has long validated these concerns. The work of Ristenpart et al. (2009), which explored information leakage in third-party compute clouds, provided an early and influential analysis of the risks associated with outsourcing data to external infrastructures.1 These threats collectively form the basis of the "trust issue" that undermines user confidence and necessitates a shift towards a more trustless, user-controlled security paradigm.

| Feature | Provider-Centric Model (e.g., Standard Google Drive) | Client-Side Encryption Model (Proposed) |
| --- | --- | --- |
| **Key Custody** | Cloud Service Provider (CSP) | User |
| **Data Access by Provider** | Technically possible; relies on operational controls and trust. | Technically impossible; provider only sees ciphertext. |
| **Data Access by Law Enforcement** | CSP can be compelled to provide decrypted data. | CSP can only provide encrypted data; keys are not held. |
| **Resilience to Provider Breach** | A breach of the CSP's key management system can expose user data. | A breach of the CSP's infrastructure does not expose plaintext data. |
| **User Control** | User controls access permissions but not the underlying keys. | User has absolute control over data and encryption keys. |
| **Integrity Verification** | Relies on CSP's internal, non-verifiable checks. | User can independently verify data integrity via client-side mechanisms. |

**Architectures for User-Controlled Data Protection: A Survey of Client-Side Encryption (CSE) Solutions**

To address the trust deficit inherent in provider-centric models, a class of solutions known as client-side encryption (CSE) has emerged. These tools and architectures aim to shift the locus of control over data confidentiality from the cloud provider to the end-user. This section surveys the principles of CSE and critically evaluates prominent existing solutions to identify the architectural and usability gaps that the proposed research seeks to fill.

**Principles of Client-Side and End-to-End Encryption**

Client-side encryption is defined by the practice of encrypting data on the user's local device *before* it is transmitted to the cloud server.19 The server, therefore, only ever receives and stores ciphertext. This process creates a "zero-knowledge" environment, where the service provider has no technical means to access the plaintext content of user files.19 This model is fundamentally different from server-side encryption, where the encryption process occurs on the provider's infrastructure, necessarily granting the provider access to both the plaintext and the encryption keys.19

The security of CSE protocols can be subjected to formal analysis, a rigorous process common in top-tier academic venues like ACM CCS for evaluating cryptographic systems against well-defined adversary models.23 While formal critiques of specific commercial CSE

*tools* are less prevalent than for communication protocols like Signal 25, the underlying cryptographic principles and design patterns can be evaluated against established security goals such as confidentiality and integrity under chosen-plaintext and chosen-ciphertext attacks.26

**Critical Evaluation of Existing CSE Tools**

Several tools have been developed to provide client-side encryption for popular cloud storage services. A critical examination of their architectures reveals significant limitations, particularly in the areas of key management, sync efficiency, and usability, which directly motivate the need for a novel approach.

* **Cryptomator**: As a leading open-source CSE tool, Cryptomator creates an encrypted "vault" within a user's cloud storage directory.22 Its architecture is notable for encrypting not only file contents but also filenames and directory structures, effectively obfuscating the metadata that could otherwise leak information.27 It achieves this by mounting a virtual filesystem, which intercepts file operations and performs on-the-fly encryption and decryption transparently for the user.28 Cryptomator's key management model is straightforward: a user-provided password is used to derive a Key-Encryption Key (KEK) via the memory-hard function

scrypt. This KEK is then used to decrypt the vault's master keys, which are stored in an encrypted masterkey.cryptomator file within the vault.28 While secure, this model's primary limitation is its fragility. It presents a single point of failure: if the user forgets their password, the data becomes permanently inaccessible.29 Although a "recovery key" can be generated, it is merely a human-readable representation of the master key itself, and the burden of its secure management falls entirely on the user.30 This design highlights the critical research gap that the proposed dissertation targets: the need for a key management system with robust and usable recovery mechanisms that do not rely solely on a user's memory or their ability to securely store a single recovery token.

* **Boxcryptor (Pre-Acquisition by Dropbox)**: Boxcryptor was a popular commercial, closed-source CSE tool that also utilized a virtual drive to encrypt files individually before synchronization.31 It offered broad compatibility with over 30 cloud providers and included advanced features like secure file sharing.33 However, its key management architecture presented a different kind of trust issue. While a "local account" option existed, the default and most common usage model required users to create a Boxcryptor account, which involved storing an encrypted version of their keys on Boxcryptor's servers.35 This model did not eliminate the need for a trusted third party but rather shifted that trust from the cloud storage provider (e.g., Google) to the CSE provider (Boxcryptor). This approach fails to achieve the "trustless" ideal where the user is the sole custodian of their keys. The acquisition of Boxcryptor by Dropbox and the subsequent discontinuation of the service for new users further underscores the vulnerability of relying on a centralized CSE provider and creates a market gap for independent, user-controlled solutions.12
* **VeraCrypt (as a CSE tool)**: VeraCrypt is a powerful and well-regarded open-source tool for full-disk encryption, but its application to cloud storage is a repurposing of its core functionality.38 The typical method involves creating a large, encrypted file container, which is then placed within a cloud synchronization folder.39 This approach suffers from a severe performance and usability drawback: it is fundamentally incompatible with the efficient, block-level synchronization mechanisms used by services like Dropbox and Google Drive. Any minor modification to a single file within the VeraCrypt container changes the entire container file, forcing the sync client to re-upload the entire multi-gigabyte file.40 This inefficiency makes it impractical for the dynamic, seamless workflow that users expect from cloud storage and which the proposed dissertation aims to preserve.

**The Security vs. Usability Trade-off**

The tension between security and usability is a central theme in the field of Human-Computer Interaction (HCI) and a frequent topic at premier academic conferences such as the Symposium on Usable Privacy and Security (SOUPS) and ACM CHI.41 The seminal 1999 study by Whitten and Tygar on PGP 5.0 famously demonstrated that a cryptographically strong system can be rendered insecure in practice if users are unable to understand and operate it correctly.45

This trade-off is particularly acute in the context of client-side encryption. The requirement for users to manage their own cryptographic keys and recovery mechanisms is a significant cognitive burden and a major usability hurdle.46 Studies on email encryption tools have consistently shown that users are overwhelmed by complex key management tasks, which can lead them to abandon the tools or adopt insecure workarounds, such as writing down passwords in plain sight.46 The proposal's emphasis on a "user-friendly key management system" is therefore not merely an enhancement but a foundational requirement for the system's overall effectiveness.

Furthermore, the design of recovery mechanisms has a profound impact on user psychology and behavior. A system that presents a single point of failure—where forgetting one password leads to catastrophic data loss—induces a fear of lockout, which is a well-documented barrier to the adoption of security technologies like two-factor authentication (2FA).47 This fear can drive users to choose weaker, more memorable passwords or to store recovery keys in insecure digital formats, thereby undermining the system's security guarantees. Consequently, providing a secure

*and* usable recovery mechanism is not a matter of convenience; it is an essential component for encouraging secure user behavior and ensuring the practical viability of the entire security model.

| Feature | Cryptomator | Boxcryptor (Legacy) | VeraCrypt (Container Method) | Proposed System (Target) |
| --- | --- | --- | --- | --- |
| **Open/Closed Source** | Open Source | Closed Source | Open Source | Open Source |
| **Key Management Model** | Password-derived; keys stored locally. | Account-based; keys stored on Boxcryptor servers (default). | Password-based; keys stored within the container file. | User-controlled; keys derived and managed locally. |
| **Key Custody** | User | Boxcryptor (default) | User | User |
| **Single Point of Failure** | Yes (forgotten password or lost recovery key). | Yes (compromise of Boxcryptor account/servers). | Yes (forgotten password or corrupted container file). | No (eliminated via threshold recovery). |
| **Recovery Mechanism** | Single, user-managed recovery key. | Password reset via Boxcryptor account. | None (password is the only key). | Robust, user-centric threshold-based recovery. |
| **Filename Encryption** | Yes | Yes (paid feature) | N/A (entire filesystem is opaque) | Yes |
| **Cloud Sync Efficiency** | High (file-level encryption). | High (file-level encryption). | Very Low (entire container re-syncs on any change). | High (file-level encryption). |
| **Target User** | Tech-savvy individuals. | Individuals and businesses. | Tech-savvy individuals. | Individuals and SMEs. |

**Cryptographic Foundations for a Trustless Overlay**

The construction of a secure client-side cryptographic overlay depends on the careful selection and correct implementation of cryptographic primitives. Each component—from the algorithm used for confidentiality to the method for deriving keys from passwords—must be chosen based on current security best practices and robust academic validation. This section details the cryptographic foundations for the proposed system, justifying each choice and highlighting improvements over the initial proposal based on a review of the state-of-the-art.

**Confidentiality via Authenticated Encryption: AES-GCM**

To ensure both confidentiality and integrity of file contents, the system will employ an Authenticated Encryption with Associated Data (AEAD) scheme. AEADs are modern cryptographic constructions that combine an encryption algorithm with a Message Authentication Code (MAC) in a secure and efficient manner.50 This integrated approach is demonstrably superior to composing encryption and authentication manually (e.g., encrypt-then-MAC), as the latter is notoriously prone to subtle implementation flaws that can lead to significant vulnerabilities.53

The chosen AEAD scheme is **AES-GCM** (Advanced Encryption Standard in Galois/Counter Mode). AES-GCM is a widely adopted industry standard, recommended by NIST and integral to modern security protocols like TLS 1.2 and 1.3.52 Its use of counter (CTR) mode for encryption allows for parallel processing of data blocks, which can yield significant performance benefits on modern multi-core processors compared to sequential modes like Cipher Block Chaining (CBC).56

The security of AES-GCM is well-established, with formal proofs demonstrating its security against adaptive chosen-ciphertext attacks (IND-CCA3), a strong security notion, provided it is implemented correctly.58 The most critical implementation requirement for AES-GCM is the uniqueness of the nonce (also known as an Initialization Vector or IV) for every encryption operation performed with the same key. Nonce reuse is catastrophic, as it can allow an attacker to recover the authentication key and subsequently forge messages, or in some cases, recover the plaintext.52 The system's design must therefore incorporate a robust mechanism for generating and managing unique nonces, especially when encrypting large files that must be processed in smaller chunks.62

**Lightweight Integrity Verification: HMAC**

While AES-GCM provides authenticated encryption for the file contents themselves, a separate mechanism is required to ensure the integrity of the overall vault structure and its metadata. For this purpose, the system will use **HMAC** (Hash-based Message Authentication Code), as specified in RFC 2104.63 HMAC is a specific construction for calculating a MAC involving a cryptographic hash function (such as SHA-256) and a secret key.64

The HMAC construction involves a two-stage hashing process that uses the key to process the message data, producing a fixed-size authentication tag.63 Its security is provably reducible to the security of the underlying hash function; if the hash function is secure (e.g., collision-resistant), then HMAC is a secure MAC.63 In the context of the proposed system, an HMAC derived from a user's master key can be used to sign a manifest file that lists the names and cryptographic hashes of all encrypted files within the vault. When the vault is unlocked, the client application can re-calculate this HMAC and verify it against the stored value. This provides a lightweight yet powerful mechanism to detect unauthorized modifications to the vault's structure on the cloud server, such as the deletion, replacement, or reordering of encrypted files—attacks that file-level AEAD alone may not prevent.66

**Key Derivation: From PBKDF2 to Argon2**

A critical component of any user-facing cryptographic system is the method used to convert a low-entropy, human-memorable password into a high-entropy, cryptographically strong key. This is the role of a Key Derivation Function (KDF).68 The initial project proposal suggests the use of PBKDF2.1 PBKDF2 is a long-standing standard that works by repeatedly applying a pseudorandom function, such as HMAC-SHA256, to the password and a salt for a specified number of iterations. This process, known as key stretching, increases the CPU time required for an attacker to perform a single password guess in a brute-force attack.69

However, the primary weakness of PBKDF2 is that its defense relies almost exclusively on computational difficulty (CPU time). In the years since its standardization, attackers have developed the capability to use massively parallel hardware, such as Graphics Processing Units (GPUs) and Application-Specific Integrated Circuits (ASICs), to dramatically accelerate password guessing attacks, significantly diminishing the effectiveness of purely CPU-bound KDFs.70

In response to this threat, a new class of memory-hard KDFs was developed. These functions are designed to require not only significant CPU time but also a large amount of RAM to compute, making them much more expensive and difficult to parallelize effectively on specialized hardware.

* **Scrypt**: An early and influential memory-hard function, scrypt was designed to raise the memory cost of password cracking, offering greater resistance to hardware-based attacks than PBKDF2.71
* **Argon2**: The winner of the multi-year public Password Hashing Competition (2013-2015), Argon2 represents the current state-of-the-art in KDF design.69 It is highly resistant to GPU and ASIC attacks and features tunable parameters for memory cost (

m), time cost (iterations, t), and parallelism (p).74 The

**Argon2id** variant is now the recommended choice as it provides a hybrid approach, offering resistance to both side-channel attacks (a strength of Argon2i) and GPU-based cracking attacks (a strength of Argon2d).74

A thorough review of the literature provides a compelling justification for enhancing the proposed system's design by replacing PBKDF2 with Argon2id. This decision aligns the project with current cryptographic best practices, demonstrates a deep engagement with the evolution of security standards, and provides a significantly stronger defense against modern offline password cracking threats.

| Feature | PBKDF2 | scrypt | Argon2id |
| --- | --- | --- | --- |
| **Primary Resistance Mechanism** | CPU Time (Iterations) | CPU Time & Memory | CPU Time, Memory & Parallelism |
| **GPU/ASIC Resistance** | Low | High | Very High |
| **Memory Hardness** | No | Yes | Yes |
| **Parameter Tunability** | Low (Iterations only) | Medium (CPU/Memory cost) | High (Time, Memory, Parallelism) |
| **Side-Channel Resistance** | N/A | Low | High (Hybrid of Argon2i/d) |
| **Current Recommendation** | Legacy | Good | Best Practice (PHC Winner) |

**The Crux of the Problem: Secure and Usable Key Management for End-Users**

The central innovation and most significant challenge identified in the dissertation proposal is the development of a novel key management system that is simultaneously secure, user-centric, and usable.1 While the cryptographic primitives discussed previously provide the necessary tools for confidentiality and integrity, their practical effectiveness hinges on a key management architecture that empowers users without overwhelming them. This section delves into the literature on user-centric key management, exploring theoretical models and practical schemes that can inform the design of a system that is resilient to both technical attacks and human error.

**The Challenge of User-Centric Key Custody**

Designing systems where non-expert users are the ultimate custodians of their own cryptographic keys is a notoriously difficult problem.77 The architecture must defend against two primary failure modes: key compromise, which leads to a breach of confidentiality, and key loss, which leads to a permanent loss of data access. The academic field of usable security, with foundational work presented at venues like SOUPS and ACM CHI, has extensively documented the difficulties users face with cryptographic concepts.44

Studies consistently show that users struggle to form accurate mental models of complex security systems like email encryption.46 When faced with intricate key management tasks, users often feel overwhelmed, leading to poor adoption rates or the use of insecure practices.46 A critical factor influencing user behavior is the presence of a single point of failure. A system where forgetting a single master password results in catastrophic and irreversible data loss creates significant user anxiety.47 This "fear of lockout" is a powerful deterrent to adoption and can motivate users to choose weak, easily remembered passwords or to store recovery information in insecure ways, thereby negating the system's security guarantees.48 Therefore, a resilient and forgiving recovery mechanism is not a peripheral feature but a core component of a usable and, by extension, secure system.

**Decentralized and User-Sovereign Key Management Models**

The proposed system's goal of ensuring keys are "private from the cloud" and user-controlled aligns with the principles of Decentralized Key Management Systems (DKMS) and the broader Self-Sovereign Identity (SSI) movement. DKMS provides a formal architectural model that aims to eliminate central authorities and single points of failure, granting users ultimate sovereignty over their keys and certificates.81 While many DKMS proposals leverage distributed ledger technology (blockchains) for sharing public key information 81, the core principles of decentralization and user control can be implemented in a more lightweight manner for a specific application. Framing the dissertation's key management system as a practical application of DKMS principles for cloud storage situates the work within a significant and contemporary trend in digital identity and security.84

**A Review of Novel Key Recovery Mechanisms**

The literature offers several cryptographic and social primitives that can be synthesized to create a robust, user-centric recovery system. The novelty of the proposed dissertation lies not in inventing a new cryptographic algorithm, but in the specific, thoughtful combination of these existing techniques into a cohesive and usable architecture that addresses the limitations of current CSE tools.

* **Cryptographic Splitting: Shamir's Secret Sharing (SSS)**: At the core of a resilient recovery system is the ability to eliminate single points of failure. Shamir's Secret Sharing (SSS) is a cryptographic algorithm that achieves this by dividing a secret (such as a master encryption key) into multiple unique parts, called "shares".87 The scheme is configured with a

k-of-n threshold, meaning the secret is split into n shares, and any k of those shares can be combined to reconstruct the original secret. Crucially, any combination of k-1 or fewer shares reveals no information about the secret.89 This mechanism directly counters the risk of a single lost password or recovery key. For example, a user could employ a 3-of-5 SSS scheme, storing one share on their primary computer, another on their mobile device, a third as a physical paper backup, and entrusting the final two shares to trusted individuals or secure locations. The loss or compromise of any two shares would not result in a loss of data access or a security breach.

* **Social Recovery**: This is a user-facing application of a threshold scheme like SSS, where the shares are distributed among a user-selected social network of "guardians".91 To recover their account, the user contacts the required threshold of guardians, who provide their respective shares to reconstruct the master key.93 While powerful, this approach introduces unique usability and security challenges. The user interface for selecting, notifying, and managing guardians must be intuitive. Furthermore, the system must be resilient to social engineering attacks, where an adversary might trick guardians into releasing their shares.92 Research from conferences such as ASIACRYPT has begun to explore formal models for social recovery, but the human factors remain a significant area for investigation.94
* **Recovery Codes**: Often presented as a mnemonic phrase or a long string of characters, a recovery code is a simpler form of backup, typically representing a single key or a single share of a secret.47 User studies conducted within the SOUPS community have shown that while conceptually simple, users frequently mismanage recovery codes. They may fail to understand their critical importance, store them insecurely (e.g., as an unencrypted screenshot on the same device they are meant to recover), or simply lose them.47 The design of the user interface during the setup process is paramount in effectively communicating the purpose and proper handling of these codes to mitigate user error.

By combining these concepts, a novel, hybrid recovery system can be designed. Such a system could use SSS as its cryptographic foundation to implement a flexible k-of-n scheme. The user interface could then allow the user to decide how to distribute these n shares, for example, keeping one as a personal recovery code, storing another on a secondary device, and entrusting a third to a social guardian. This approach provides redundancy and respects user autonomy, directly addressing the critical gap in existing CSE tools.

| Mechanism | Description | Security Assumptions | Usability Challenges | Mitigation Strategy in Proposed System |
| --- | --- | --- | --- | --- |
| **Single Master Password** | A single password derives the master key. | User remembers the password; password has sufficient entropy. | High risk of permanent data loss if forgotten (single point of failure). High user anxiety. | Used only as the primary access method, not the sole key. Backed by a robust recovery system. |
| **Recovery Code** | A single, static token (e.g., mnemonic phrase) that can restore access. | User stores the code securely and has access to it when needed. | Users often store it insecurely, misunderstand its importance, or lose it. | Can be implemented as one share in a threshold scheme, reducing its criticality. UI must emphasize secure offline storage. |
| **Shamir's Secret Sharing (Multi-device)** | Secret is split into k-of-n shares, stored on different user-owned devices. | An attacker cannot compromise k devices simultaneously. User does not lose more than n-k devices. | Requires user to own and manage multiple devices. Synchronization and management can be complex. | Offer as an option for users to distribute shares across their personal devices (laptop, phone, backup drive). |
| **Social Recovery** | Secret shares are distributed to trusted "guardians" in the user's social network. | Guardians are trustworthy and will not collude. An attacker cannot compromise k guardians. | Selecting and managing guardians can be socially awkward or complex. Guardians may lose shares or become unavailable. | Offer as an optional recovery method. The UI must guide users in selecting guardians and clearly explain the process and risks involved. |

**Synthesis: Identifying the Research Gap and Justifying the Proposed Approach**

The preceding review of the literature on public cloud storage security, client-side encryption tools, and key management systems reveals a clear and compelling justification for the proposed dissertation. While individual components of a secure, user-centric cloud storage solution exist in isolation, their holistic and usable integration remains an unsolved problem and represents a significant research gap.

A synthesis of the state-of-the-art shows that the landscape is defined by a fundamental trade-off. On one hand, mainstream public cloud storage providers like Google Drive and Dropbox offer highly usable and scalable services but operate on a provider-centric, trusted custodian model. This model, by design, fails to provide users with true zero-knowledge privacy and exposes them to risks from internal threats, external breaches of the provider's core systems, and legal compulsion.1

On the other hand, existing third-party client-side encryption tools that attempt to remedy this trust deficit introduce their own critical flaws. Some, like the legacy model of Boxcryptor, merely shift the locus of trust to another centralized entity, failing to deliver genuine user sovereignty over keys.35 Others, like Cryptomator and VeraCrypt, while offering true user-controlled encryption, create a fragile security model with a single point of failure: the user's master password.28 The consequence of forgetting this single credential is the irreversible loss of all encrypted data. As established by extensive research in usable security, this high-stakes failure mode creates significant user anxiety, acting as a major barrier to adoption and encouraging insecure user behaviors.46

Therefore, the literature delineates a distinct and multi-faceted research gap: there is a lack of a practical, end-to-end client-side encryption framework that successfully integrates three essential properties:

1. **Strong, Modern Cryptography**: The consistent application of state-of-the-art, peer-reviewed cryptographic primitives, including an AEAD scheme like AES-GCM for confidentiality and integrity, and a memory-hard key derivation function like Argon2id to protect against modern brute-force attacks.
2. **True User Sovereignty**: A key management architecture founded on decentralized principles, ensuring the user is the sole and absolute custodian of their cryptographic keys, which are never exposed to the cloud provider or any other third party.
3. **Resilient and Usable Recovery**: A robust and intuitive key recovery mechanism, built on a foundation of threshold cryptography such as Shamir's Secret Sharing, which eliminates single points of failure and allows users to recover from the loss of a primary credential without compromising the system's zero-knowledge guarantee.

The dissertation project outlined in the proposal is precisely designed to address this holistic gap.1 It moves beyond the theoretical exploration of DKMS or the isolated implementation of CSE tools. By aiming to design, prototype, and evaluate a system that synthesizes these three critical elements, the research will make a tangible contribution to the field. The final proof-of-concept will serve as a practical framework and a clear demonstration of how to engineer a more "trustless" cloud storage solution that is both cryptographically sound and genuinely usable for its intended audience of individuals and SMEs. This endeavor directly answers the primary research question of how to effectively design such a system while balancing security, usability, and robust key management, thereby providing a foundational step towards empowering users with absolute control over their data's privacy in the cloud.