ISSN: 2581-1274

Design and Implementation of a Cryptography Key Management System API Using SoftHSM and PKCS#11

Rivaldo Hardiansyah [a,1,*], Nurhasanah [a,2], Fadly Ariadi [a,3]

[a] *University of Pamulang, Jl. Raya Puspitek, South Tangerang 15310, Indonesia*
[1] *rivaldoh88@gmail.com*;* [2] *dosen01123@unpam.ac.id;* [3] *dosen01123@unpam.ac.id*
*Corresponding author

ARTICLE INFO

Article history:
Published
January 6, 2026

Keywords:
Cryptography
Key Management
SoftHSM
PKCS#11
API

ABSTRACT

This paper presents the design and implementation of a Cryptography Key Management System (KMS) API that leverages SoftHSM as secure key storage and the PKCS#11 (Cryptoki) standard as the cryptographic interface. Motivated by the need to protect sensitive organizational data and to comply with regulations, the system centralizes key lifecycle operations—generation, storage, use for AES-based encryption/decryption, and key destruction—so that key materials never leave the HSM. The study follows a waterfall-style development process comprising requirements analysis, system design, implementation, testing, and evaluation. Results show the API correctly performs cryptographic operations with keys resident in the HSM and supports basic auditing of operations. The approach demonstrates a practical, low-cost alternative to physical HSMs for small-to-medium environments while aligning with information security management practices.

I. Introduction

Modern organizations increasingly rely on cryptography to protect sensitive data at rest and in transit. However, the security of a cryptographic system depends not only on strong algorithms, but also on how cryptographic keys are created, stored, and used. In many deployments, application code still embeds keys in configuration files or databases, exposing them to accidental leakage and unauthorized access. A practical approach is to centralize key custody and restrict key use through a dedicated Key Management System (KMS) that applications can access over well-defined interfaces.

Although prior research has proposed various software-based key-management approaches, several limitations remain. Many implementations store symmetric keys in encrypted files or database tables, which—despite encryption—still allow key material to be exported or accessed by privileged system users. Other studies focus only on key storage or key generation without enforcing hardware-backed isolation, allowing application components to handle raw keys during cryptographic operations. Several SoftHSM-based solutions demonstrated key protection concepts but did not expose an application-level API, reducing their practicality for integration with external systems. These gaps motivate the need for an accessible, programmable, and strictly non-exportable key-management mechanism.

In contrast, modern KMS deployments in industry—such as AWS KMS, Google Cloud KMS, Azure Key Vault, and HashiCorp Vault—embed strong isolation guarantees by relying on FIPS-validated Hardware Security Modules (HSMs), enforcing non-exportable keys, automated key-rotation policies, granular role-based authorization, and full audit trails. Emerging trends also incorporate confidential-computing enclaves and zero-trust architectures to minimize the attack surface for key access. However, these platforms typically require cloud connectivity, subscription costs, or dedicated hardware, making them less practical for small organizations or on-premises

environments that need strong protections without the complexity of enterprise-grade KMS infrastructures.

This paper presents a KMS Application Programming Interface (API) that integrates a software-based Hardware Security Module (SoftHSM) via the PKCS#11 standard (Cryptoki). The objective is to ensure that symmetric keys—specifically AES-256 keys—are generated, stored, and used inside the HSM boundary, while applications reference them only by labels. In this model, key material never leaves the HSM, and cryptographic operations (encryption and decryption) are invoked through PKCS#11 mechanisms. The API is implemented with a Django backend and a simple web front end, making it suitable for on-premises or standalone deployments.

The scope of this work focuses on foundational key-management capabilities: secure key custody and controlled key use. The system demonstrates key generation inside the HSM, label-based key selection, and AES encryption/decryption where plaintext and ciphertext flow through the service while secret keys remain non-exportable. Comprehensive enterprise features—such as automated key rotation policies, granular role-based authorization, and extensive audit trails—are acknowledged as future work.

The contributions are threefold: (1) a practical, low-cost KMS architecture using SoftHSM and PKCS#11; (2) an implementation of a RESTful API that enforces "keys never leave the HSM" through label-based invocation; and (3) functional validation via black-box tests showing that data encrypted with a labeled key can be correctly decrypted, thus confirming correct custody and use. This foundation illustrates a viable path for organizations seeking stronger key protection without the cost of physical HSMs.

II. The Proposed Method/Algorithm

This study proposes a Key Management System (KMS) API that integrates a software-based hardware security module (SoftHSM) via the PKCS#11 interface to enforce secure key custody and controlled key use. Applications never handle raw key material; instead, they reference keys by label and invoke AES-256 encryption and decryption through REST endpoints implemented in Django. Keys remain resident and non-exportable within the HSM boundary, while only operation metadata—such as key label, request identifier, timestamp, and execution outcome—is persisted in the relational store (MySQL). This architecture shifts confidentiality dependence away from application-held secrets and toward HSM-guarded key operations, making the system practical for on-premises or standalone deployments.

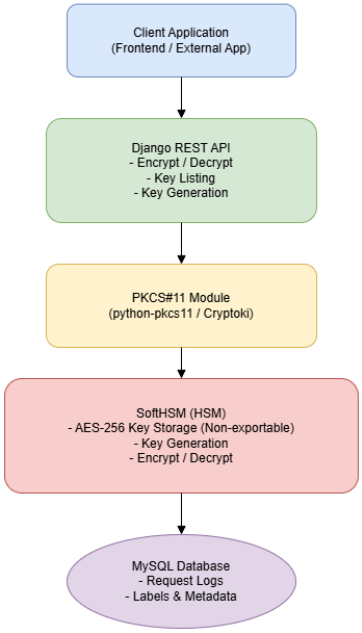


Fig. 1. System architecture depicting the interaction among the KMS API, the PKCS#11 interface, the SoftHSM cryptographic module, and the metadata database.

Figure 1 illustrates the overall system architecture, consisting of four interacting layers: the REST API, the PKCS#11 interface, the SoftHSM environment, and the metadata database. Client applications communicate with the Django API through HTTPS calls. The API then interacts with the PKCS#11 module to open authenticated sessions, locate keys by label, and execute cryptographic operations inside the HSM boundary. SoftHSM stores AES-256 keys as non-exportable objects and exposes cryptographic mechanisms through PKCS#11, ensuring that plaintext and ciphertext transit through the API while the secret key material never leaves the secure module. The MySQL database records only non-sensitive metadata for traceability and auditing.

Operationally, each encryption or decryption request triggers a PKCS#11 session authenticated with a user PIN. Once authenticated, the module locates an AES-256 key by its assigned label and performs the requested operation using a fresh IV or nonce for every call. AES-GCM is preferred due to its built-in authenticity protection, while AES-CBC with PKCS#7 padding is supported as a compatibility fallback. Key generation is performed during initialization or as needed and uses PKCS#11 attributes that explicitly prevent key extraction. Read-only key listing returns only labels rather than raw key bytes, and key deletion is governed by policy checks and logged for auditability.

Functional validation follows a black-box testing approach. A ciphertext produced through the API using a selected key label is successfully decrypted back to its original plaintext, demonstrating correct binding between label-based requests and the internal key objects stored within the HSM. This confirms that keys never leave the SoftHSM boundary and that the proposed API enforces secure, controlled, and non-exportable key use throughout all operations.

III. Method

A. System Analysis

This application aims to assist users in encrypting and decrypting text and files by referencing a pre-provisioned 256-bit symmetric key (AES-256) identified by a label in SoftHSM. Users provide the target data (plaintext or ciphertext) and the intended key label; the system then invokes a REST API that authenticates to SoftHSM via the PKCS#11 interface, resolves the label to the corresponding non-exportable key, and performs the requested operation inside the HSM. The API returns the resulting ciphertext or plaintext to the user for retrieval and safe storage. The proposed method—SoftHSM-backed, label-based AES-256 accessed through PKCS#11—offers advantages in safeguarding key material (keys never leave the HSM), reducing the risk of application-side leakage, enabling auditable and repeatable operations through label-centric logs, and remaining practical and cost-effective for on-premises or standalone deployments.

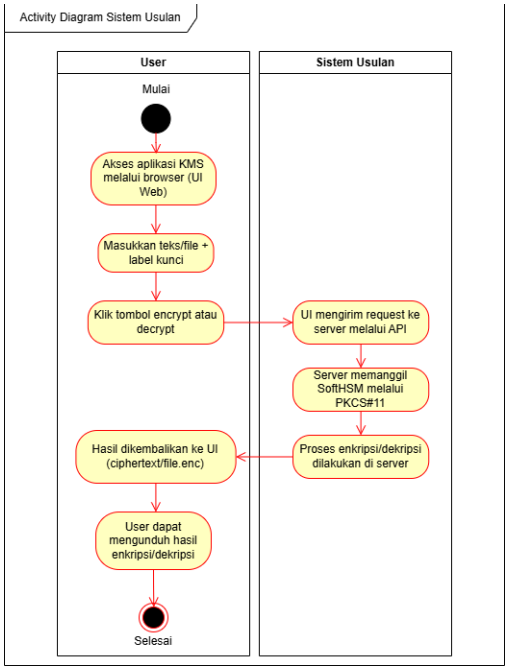


Fig. 2. Analysis of the Proposed Method

B. Unified Modelling Language (UML)

Unified Modeling Language (UML) is a standardized notation for specifying, visualizing, and documenting software-intensive systems. UML separates concerns by providing diagram types that capture different views—behavioral, structural, and interaction—thereby improving shared understanding among stakeholders.

a) Use Case Diagram

Expresses the system's external functionality from the user's perspective by showing actors and the services (use cases) they invoke.

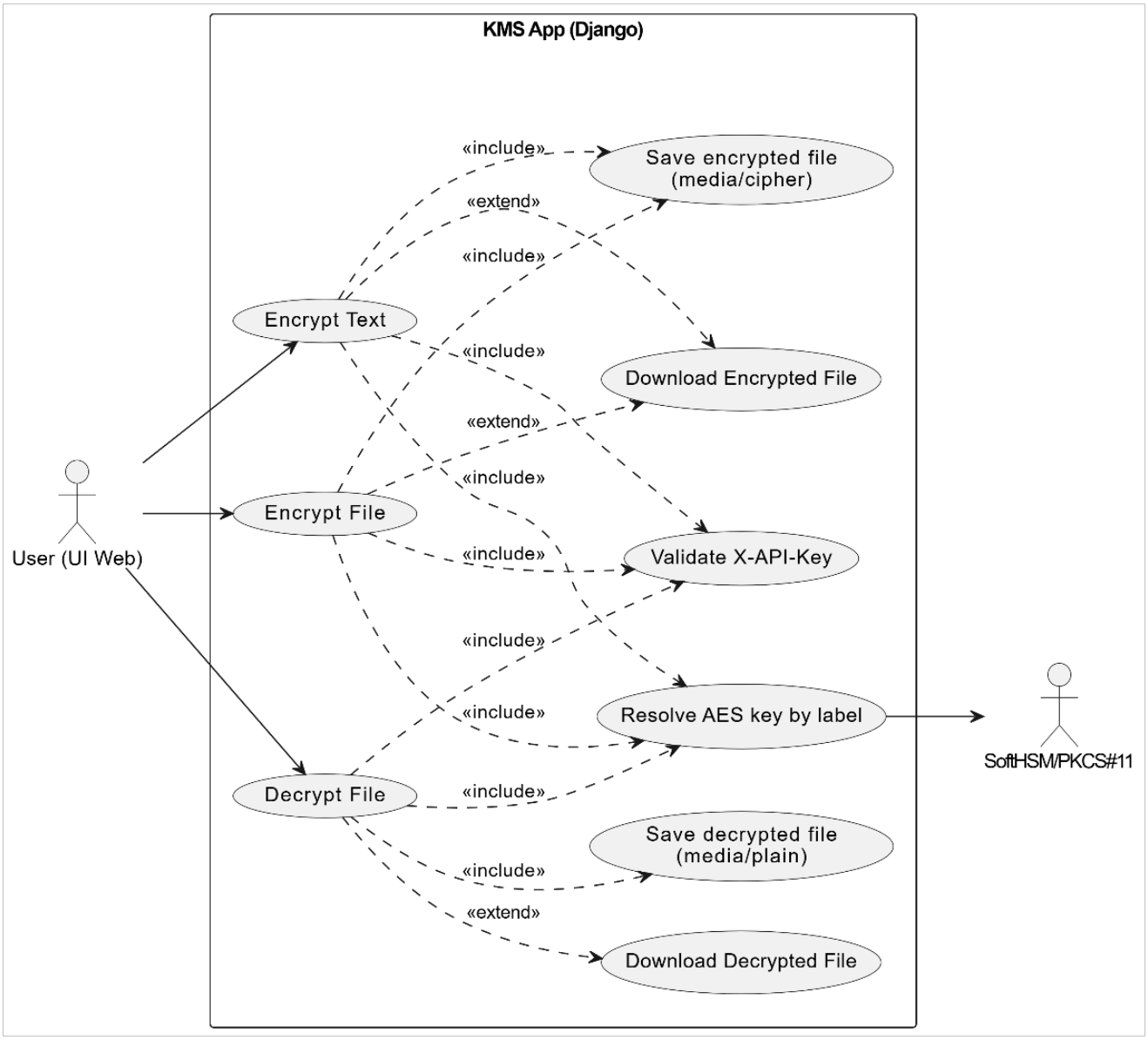


Fig. 3. Use Case Diagram "User" And "Admin (SoftHSMPKCS#11)"

Description: This section involves users and *admin* who have different access levels to each menu in the application.

b) Sequence Diagram

Depicts time-ordered interactions between components/actors via messages, emphasizing call order and lifelines.

- Sequence Diagram “Encrypt Text.”

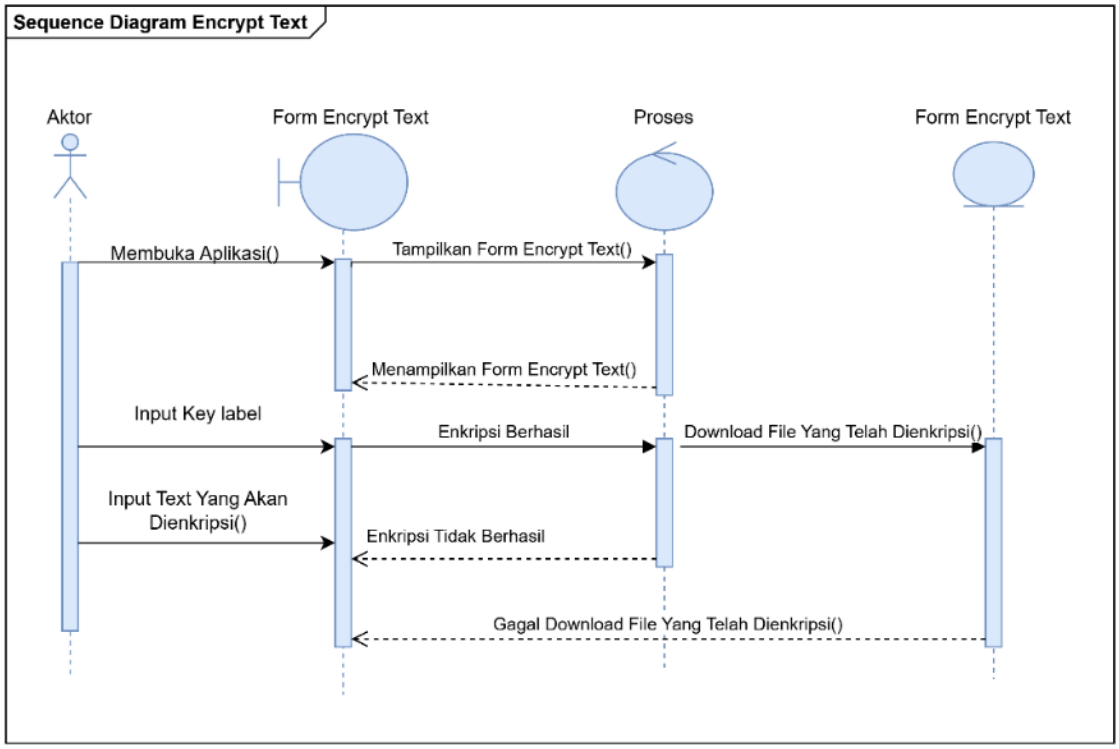


Fig. 4. Sequence Diagram "Encrypt Text"

Description: User opens the application and inputs the text to be encrypted..

- Sequence Diagram “*Encrypt File*”

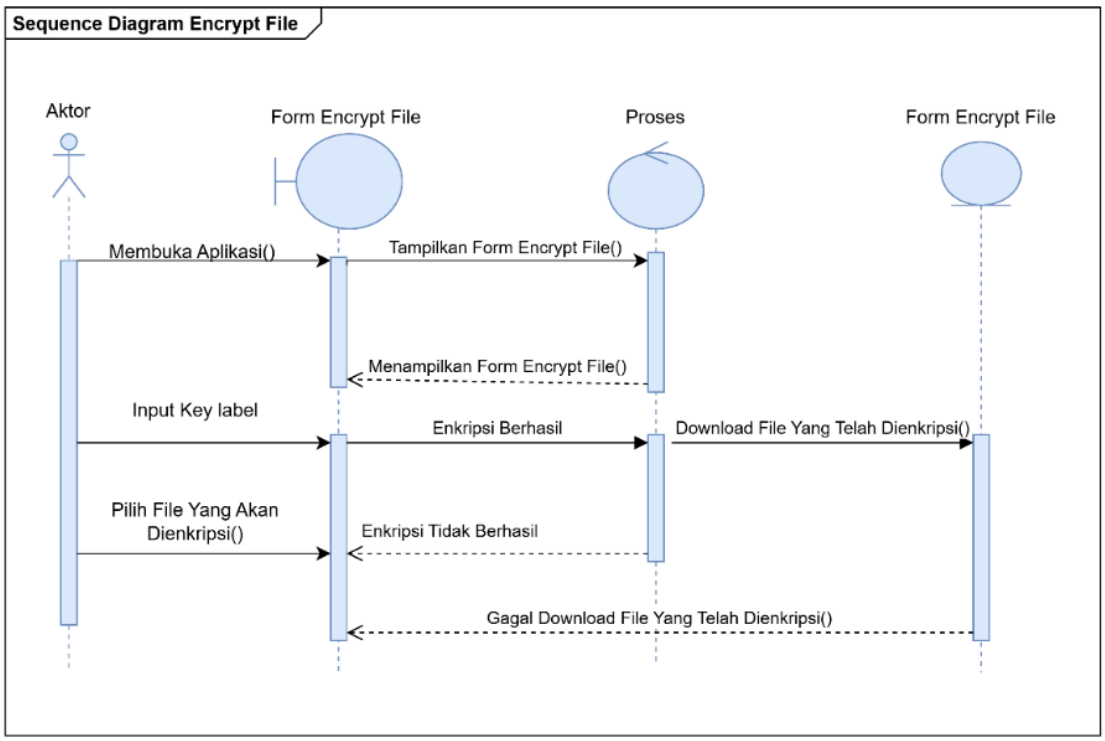


Fig. 5. Sequence Diagram "Encrypt File"

Description: User opens the application and uploads the file to be encrypted.

- Sequence Diagram “Decrypt File”

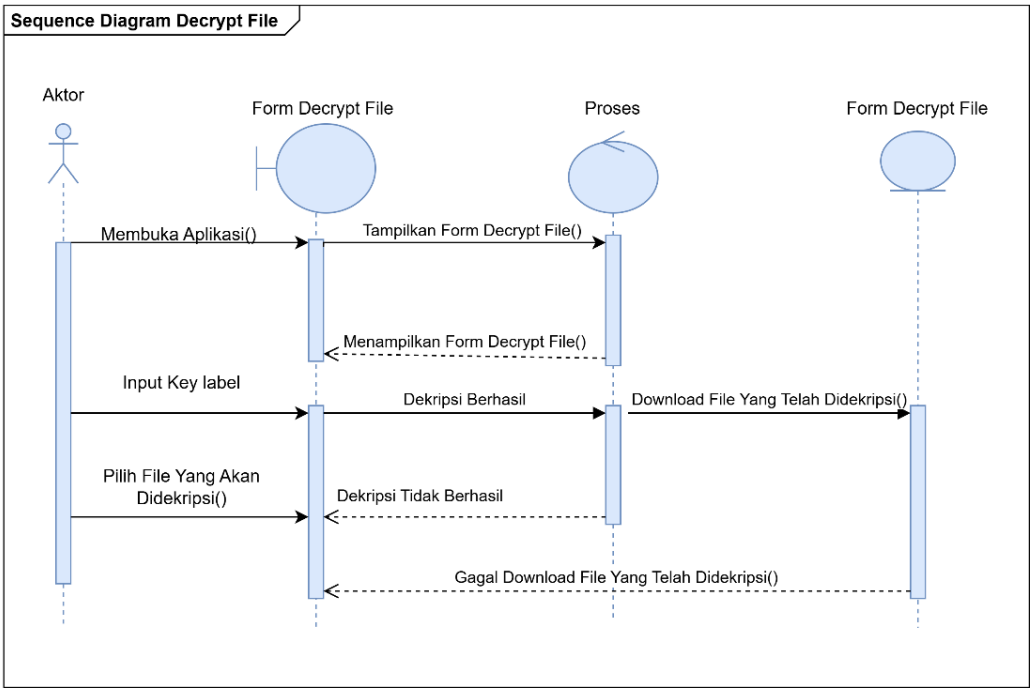


Fig. 6. Sequence Diagram "Decrypt File"

Description: User opens the application and uploads the file to be decrypted.

c) Class Diagram

Represents the static structure—classes, attributes, operations, and relationships (associations, generalization, composition)—that underpins the system’s data and logic.

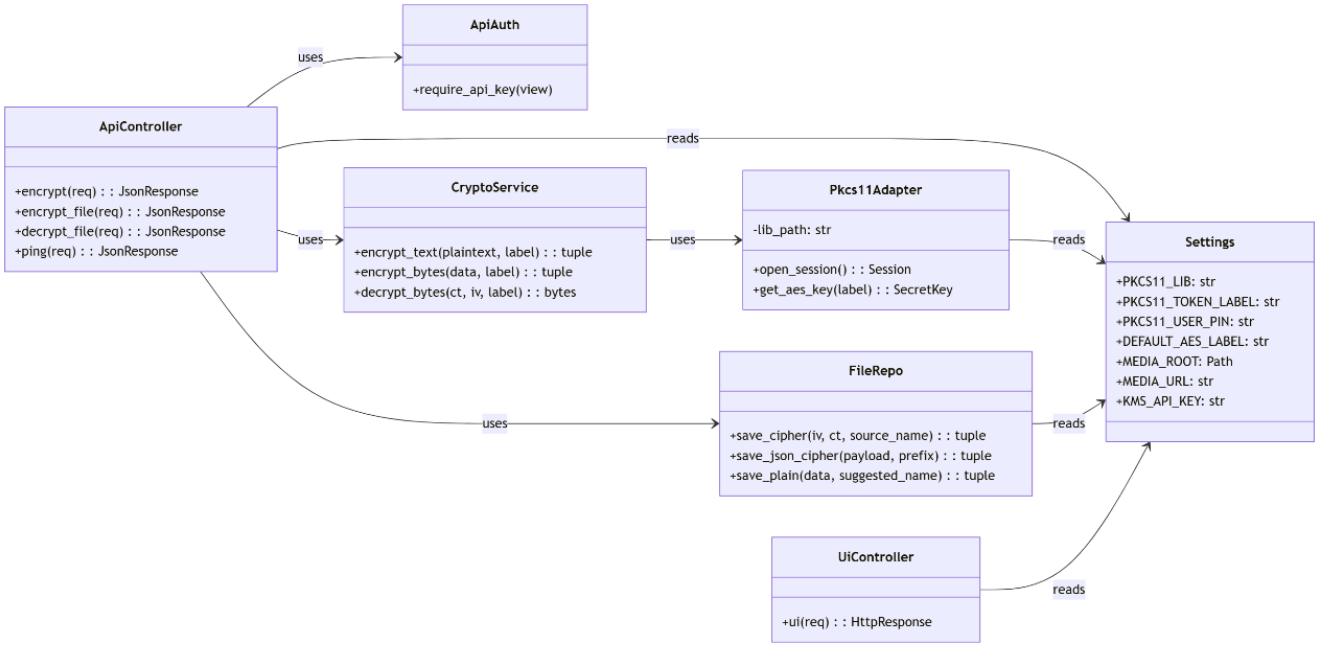


Fig. 7. Class Diagram

IV. Results and Discussion

A. Implementation

The system is implemented as a Django-based REST service that integrates SoftHSM through the PKCS#11 library. At runtime, the service opens a PKCS#11 session with a user PIN, looks up the requested key by label, and executes AES-256 encryption or decryption within the HSM boundary, key objects are non-exportable and never leave SoftHSM. MySQL is used solely as a server-side session store (Django sessions) and lightweight operational metadata, it does not persist cryptographic keys, plaintext, or ciphertext. Results are returned directly to the caller, and only minimal request context (e.g., timestamp, request identifier, operation outcome) may be recorded for traceability. This configuration keeps the application largely stateless with respect to sensitive data while preserving the "keys-never-leave-the-HSM" property enforced by PKCS#11.

1) User Interface Implementation

The prototype provides a single-panel main interface. Users enter a key label, choose the operation (encrypt/decrypt), and work with either text or files, results are returned for immediate download. No login page is included.

a) Main Interface

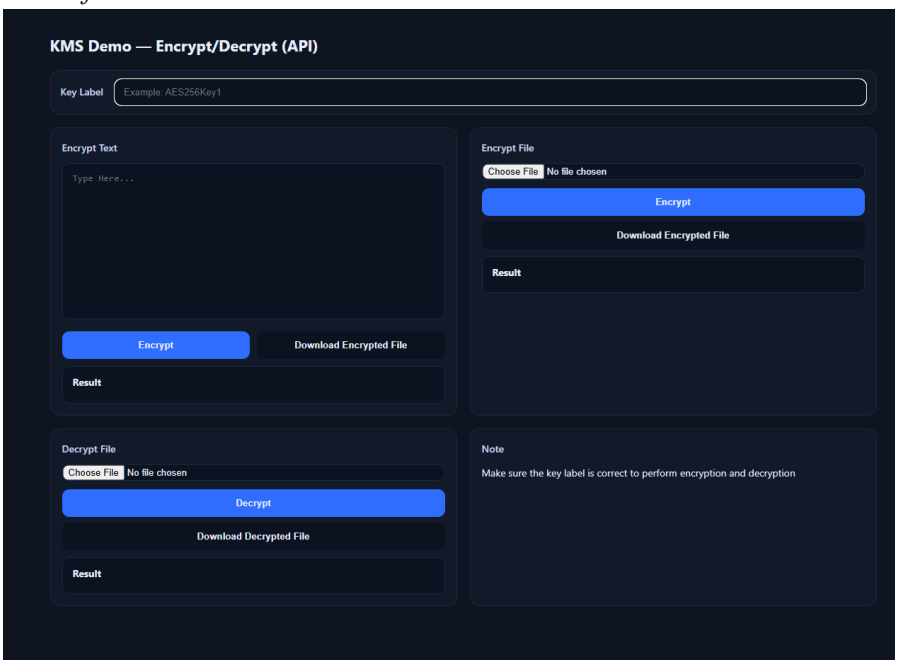


Fig. 8. Main Interface

Description: A single-panel interface where users input a key label to encrypt text or a file and to decrypt a file, keys remain non-exportable inside SoftHSM (PKCS#11).

2) Encrypt Text.

Users enter a key label and plaintext, then select Encrypt. The service authenticates to SoftHSM, locates the labeled AES-256 key, applies a fresh IV/nonce per call (AES-GCM by default or AES-CBC with PKCS#7 padding as fallback), and returns the ciphertext for download or copy.

3) Encrypt File.

Users enter a key label, choose a file, and click Encrypt. The server performs the encryption within the HSM and returns the resulting ciphertext file. No plaintext or key bytes are persisted; only the downloadable result is provided to the user.

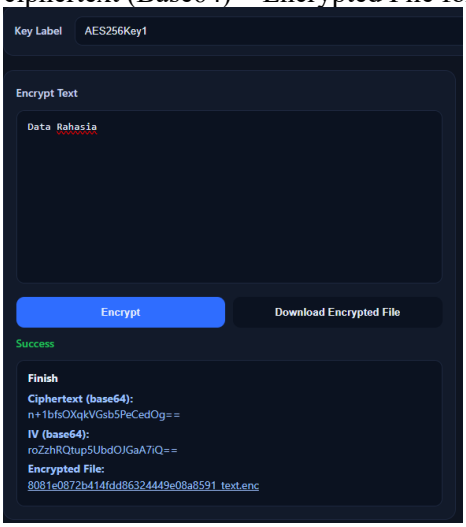
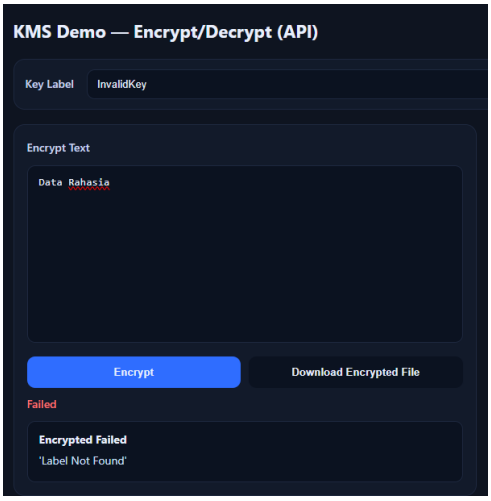
4) Decrypt File.

Users enter a key label, select the ciphertext file produced by the system, and click Decrypt. The server validates the metadata (mode, IV/nonce, tag if applicable), uses the labeled key inside SoftHSM, and returns the recovered plaintext file. A successful round-trip confirms correct label-based custody and use.

B. Testing

Black box testing focuses on the functional aspects of the system, ensuring it meets user requirements by evaluating integrated components. It assesses system correctness based solely on outputs from specified inputs, without examining internal processes. This testing verifies alignment with specified functions.

Table 1. Blackbox Testing Encrypt Text

Test Case	Scenario	Expected Result	Actual Result
Encrypt text with valid labels	Input text = "Data Rahasia", Label = "AES256Key1"	The text is successfully encrypted and displayed in ciphertext (Base64) + Encrypted File form 	Pass
Encrypt text with invalid labels	Input text = "Data Rahasia", Label = "InvalidKey"	The system displays the message "Failed" "Label Not Found" 	Pass

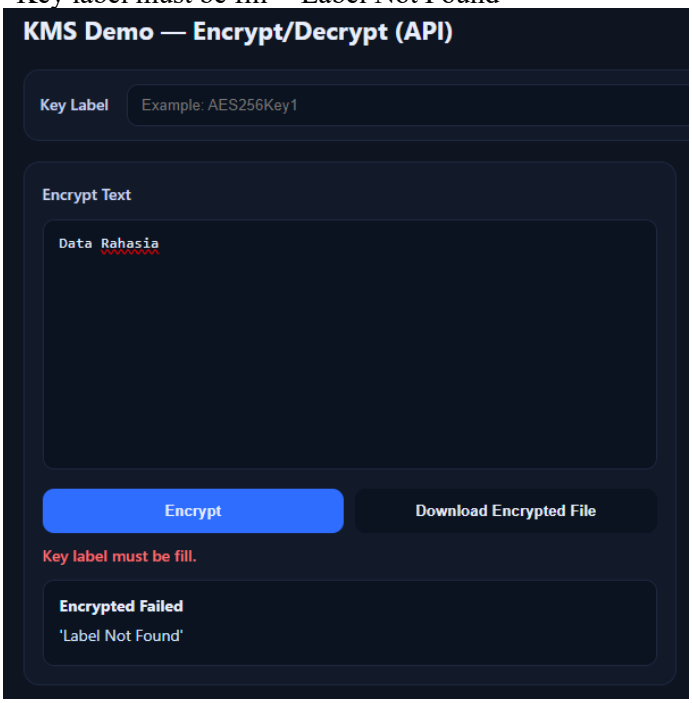
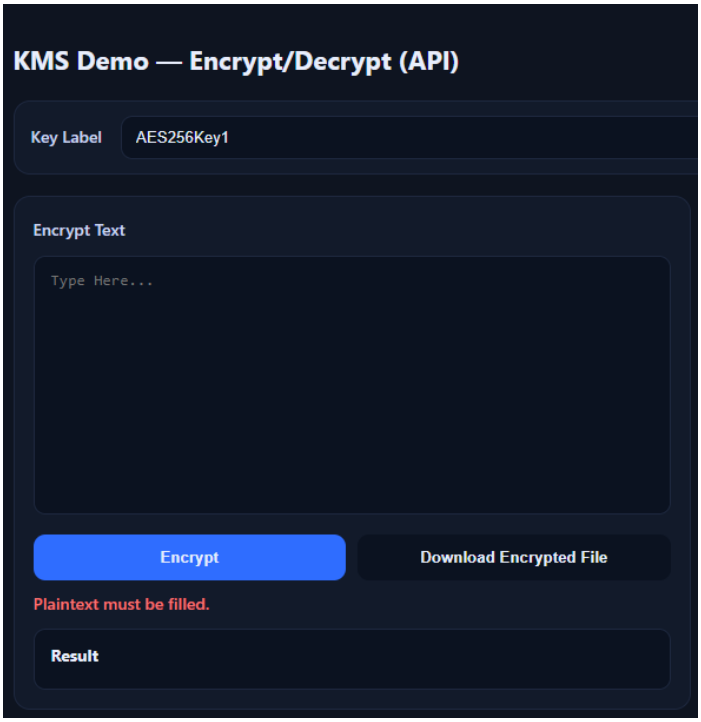
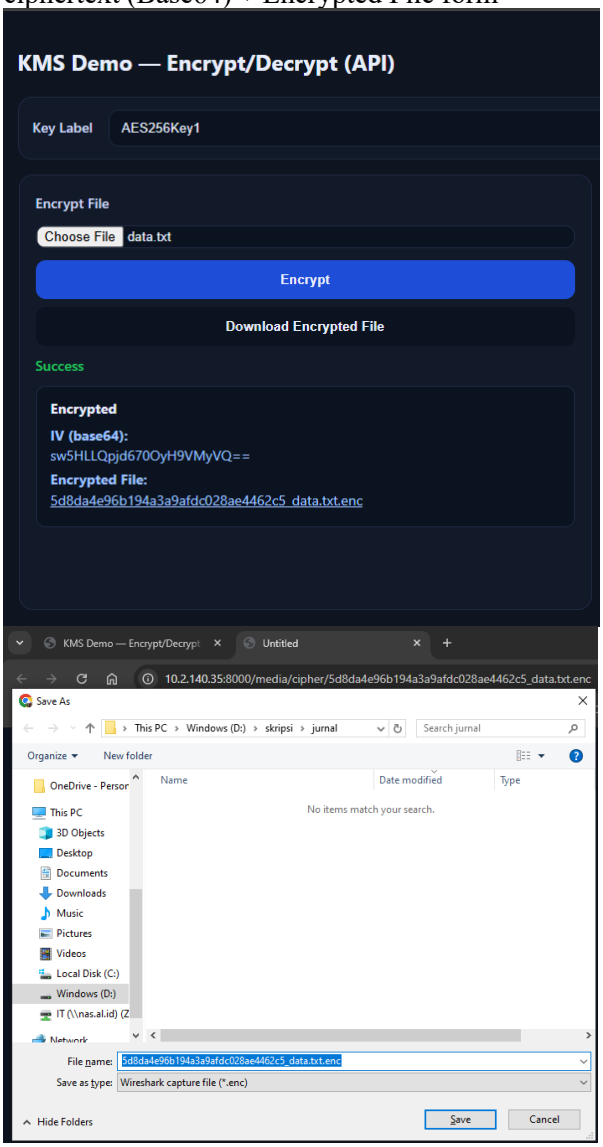
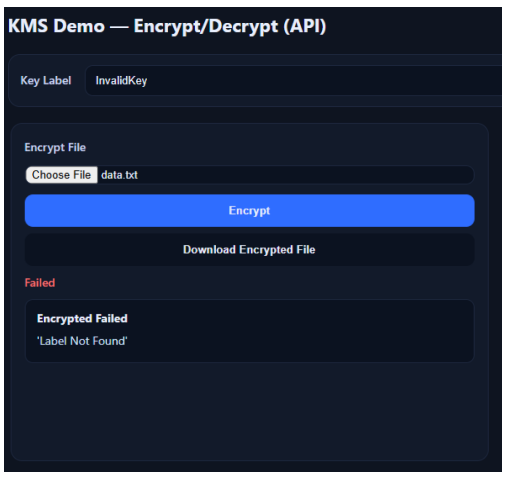
Encrypt text with empty labels	Input text = “Data Rahasia”, Label = “”	The system rejects the process and displays the message “Key label must be fill” “Label Not Found” 	Pass
Encrypt text without text content	Input text = “” Label = “AES256Key1”	The system displays a warning message “Plaintext must be filled” 	Pass

Table 2. Blackbox Testing Encrypt File

Test Case	Scenario	Expected Result	Actual Result

Encrypt file with valid labels	File = data.txt, Label = "AES256Key1"	The file is successfully encrypted and displayed in ciphertext (Base64) + Encrypted File form 	Pass
Encrypt file with invalid labels	File = data.txt, Label = "InvalidKey"	The system displays the message "Failed" "Label Not Found" 	Pass

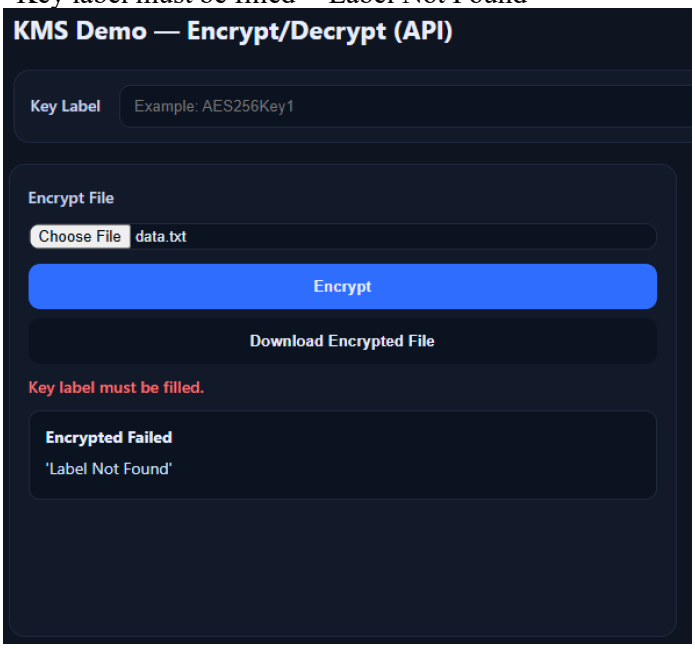
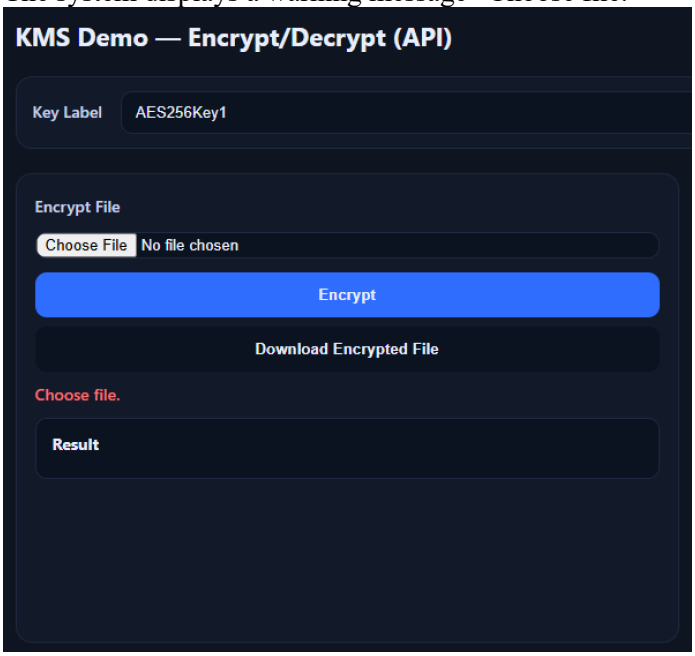
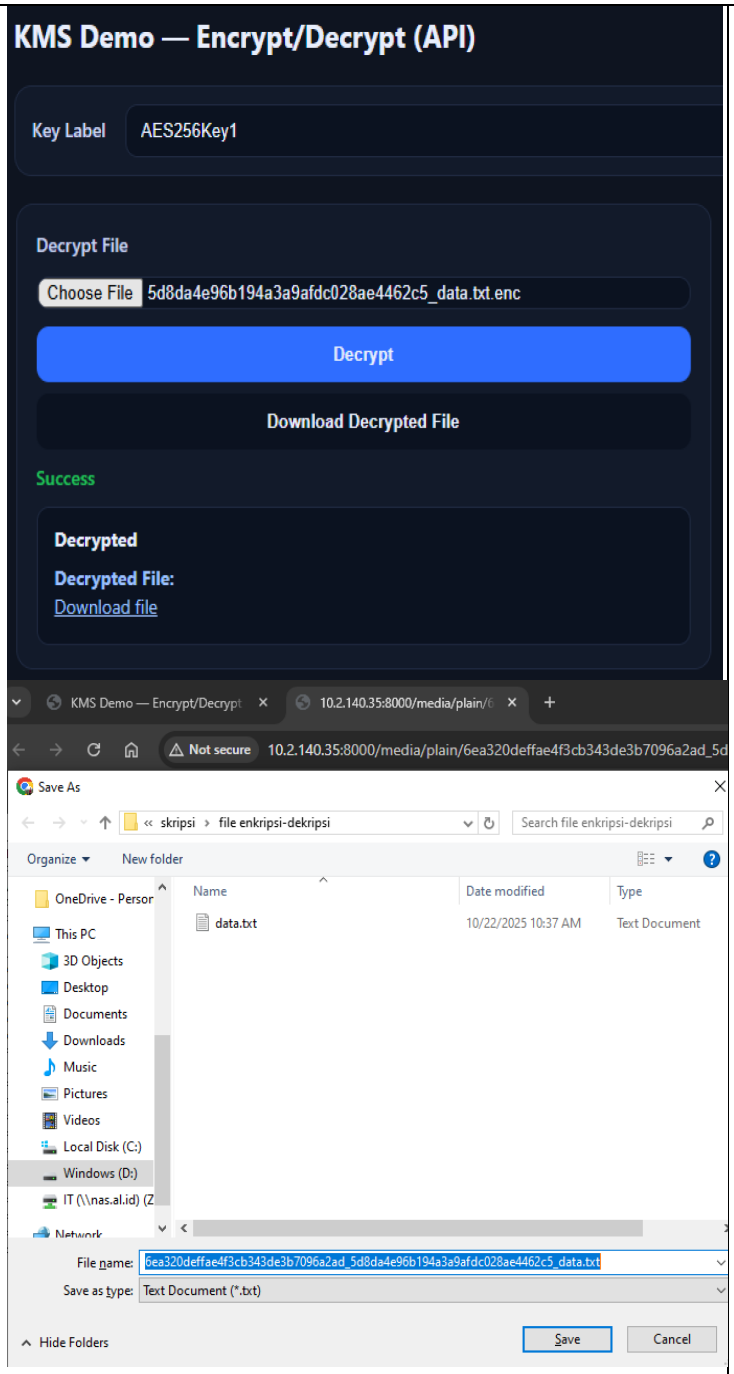
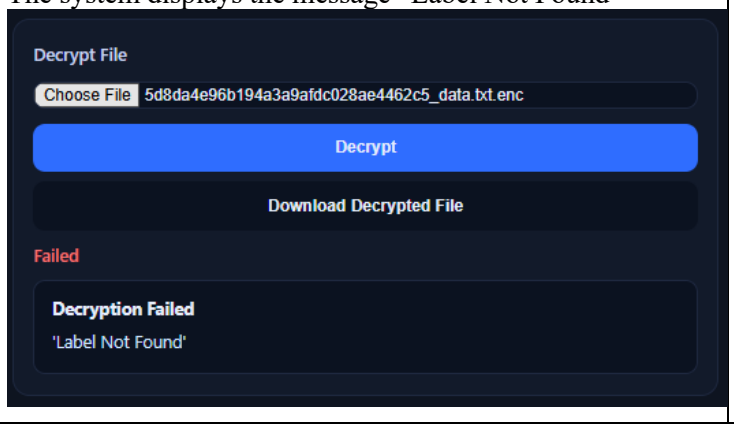
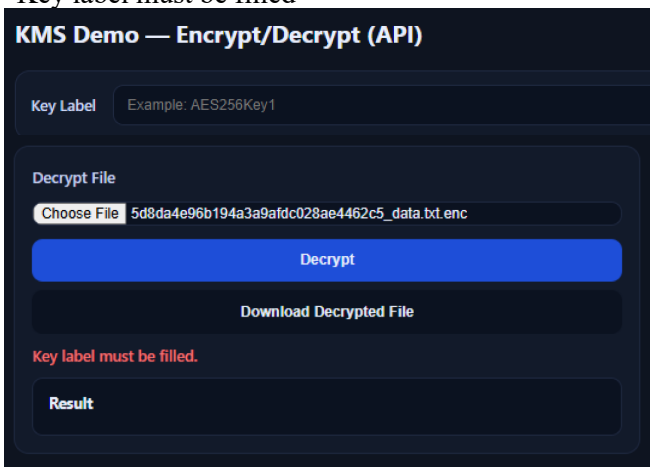
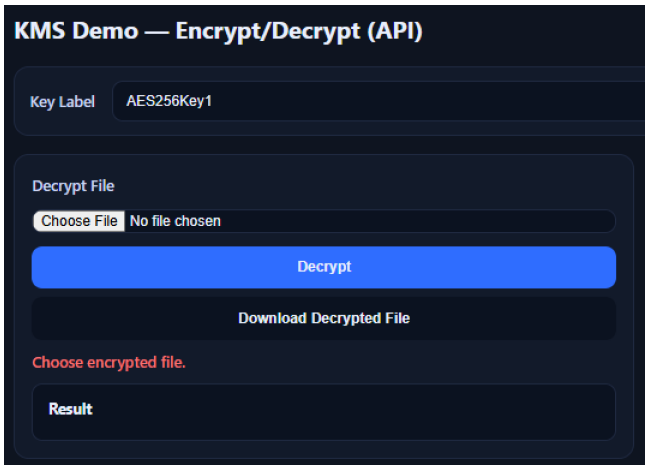
Encrypt file with empty labels	File = data.txt, Label = “”	The system rejects the process and displays the message “Key label must be filled” “Label Not Found” 	Pass
Encrypt file without file	File not choose, Label = “AES256Key1”	The system displays a warning message “Choose file.” 	Pass

Table 3. Blackbox Testing Decrypt File

Test Case	Scenario	Expected Result	Actual Result
Decrypt file with	File = data.txt.enc, Label =	The file was successfully restored to data.txt and the Download Decrypted File button is working.	Pass

valid labels	"AES256Key1"		
Decrypt file with invalid labels	File = data.txt.enc, Label = "InvalidKey"	The system displays the message "Label Not Found" 	Pass

Decrypt file with empty labels	File = data.txt.enc, Label = “”	The system rejects the process and displays the message “Key label must be filled” 	Pass
Decrypt file without file	File not choose, Label = “AES256Key1”	The system displays a warning message “Choose encrypted file” 	Pass

Based on testing results, the KMS API performs successfully across all evaluated scenarios. It operates efficiently and meets the specified functional requirements with reliable and accurate outputs, while ensuring that cryptographic keys remain non-exportable inside SoftHSM.

V. Conclusion

This paper presented the design and implementation of a label-based Key Management System (KMS) API that integrates SoftHSM through the PKCS#11 interface. The prototype, built with a Django REST backend and using MySQL only for server-side sessions, ensures that AES-256 keys are generated, stored, and used inside the HSM boundary and never exported. Black-box tests covering text encryption, file encryption, and file decryption yielded the expected outputs with a full pass rate, and the integrity check confirmed that decrypted files match their originals. Error scenarios (e.g., invalid or empty labels and missing files) were handled appropriately without leaking internal details. These results indicate that the system meets its functional requirements and demonstrates a practical, low-cost approach to improving key custody by shifting confidentiality dependence from application-held secrets to HSM-guarded keys.

Compared with traditional application-level key management, where symmetric keys are commonly embedded in configuration files, environment variables, or database entries, the proposed KMS API provides several clear advantages. Secret keys remain non-exportable inside the HSM,

reducing the risk of leakage through source code exposure, server misconfiguration, or credential compromise. Applications reference keys only by labels, avoiding direct handling of sensitive material and enabling centralized control over key use. The architecture also supports auditable and policy-driven operations, offering stronger security guarantees than decentralized key storage practices. These advantages highlight the value of a structured, HSM-backed KMS model for organizations seeking to enhance data protection without deploying physical HSM hardware.

References

[1] A. V. Chandra Christian, "Implementation of a Web-based Car Wash Queue Application Using Php, Javascript, Html, Css, and Uml," *Jati (Journal of Information Technology Students)*, Apr. 2024.

[2] S. Dewi, S. P. Adithama, and A. T. Suhardi, "Testing the Doctor to Doctor Application Using the Black Box Testing Method," *Konstelasi: Convergence of Technology and Information Systems*, vol. 3, no. 1, Jun. 2023.

[3] Siska Narulita, Ahmad Nugroho, and M. Zakki Abdillah, "Unified Modeling Language (Uml) Diagram for the Design of a Research and Community Service Management Information System," *Bridge :Journal of Information Systems and Telecommunications*, vol. 2, no. 3, pp. 244–256, Aug. 2024, doi: 10.62951/bridge.v2i3.174.

[4] U. Patkar, P. Singh, H. Panse, S. Bhavsar, and C. Pandey, "Python for Web Development," *International Journal of Computer Science and Mobile Computing*, vol. 11, no. 4, pp. 36–48, Apr. 2022, doi: 10.47760/ijcsmc.2022.v11i04.006.

[5] G. Skoglund, "Use of the Kmip Protocol for Pki Applications," Aug. 2024.

[6] J. T. Amael, J. E. Istiyanto, and O. Natan, "Enhancing Industrial Cybersecurity: Softhsm Implementation on Sbcs for Mitigating Mitm Attacks," Sep. 2024, [Online]. Available: http://arxiv.org/abs/2409.09948

[7] K. Khairani and M. Z. Siambaton, "Securing Text Data Using Elgamal and Xor Cryptographic Algorithms From Hacker Attacks," *Sudo Journal of Information Technology*, vol. 2, no. 4, pp. 176–187, Dec. 2023, doi: 10.56211/sudo.v2i4.401.

[8] V. Jain, "A Review on Different Types of Cryptography Techniques," *Academicia: An International Multidisciplinary Research Journal*, vol. 11, no. 11, pp. 1087–1094, Nov. 2021, doi: 10.5958/2249-7137.2021.02568.4.

[9] D. Widyawan and Imelda, "File Security Using Cryptography With the Web-based Aes-128 Method at the National Transportation Safety Committee," 2021.

[10] J. Soebagyo and I. Kurniawan, "Implementation of Key Matrix Algorithms for Academic Data Security," 2020.

[11] H. D. Novianti and Ahmad Tri Hidayat, "IMPLEMENTASI KRIPTOGRAFI ADVANCED ENCRYPTION STANDARD 128 BIT DALAM PENGAMANAN DATA KEUANGAN KAS," *Jurnal Komputer dan Teknologi*, pp. 27–34, Jan. 2023, doi: 10.58290/jukomtek.v1i2.51.

[12] B. Olivia Putri Irine Irawan *et al.*, "Cryptographic Implementation in Data Security Using Advanced Encryption Standard (Aes) Algorithm," vol. 11, no. 2, 2023.

[13] R. Pramitasari and I. Rofiki, "Elgamal Public Key Cryptanalysis Using Ridge Neural Networks Polynomial," *Amik Bsi Computer Engineering Journal*, vol. 8, no. 2, Jul. 2022, doi: 10.31294/jtk.v4i2.

[14] Rahmaniah, M. Firman Aditya, W. Arfanda, and V. Ndika purnama, "Study of Symmetric Key Cryptography Algorithms in Data Security Using the Comparison Method," *Siteba Journal*, vol. 2, no. 1, p. 2023, 2023, [Online]. Available: https://journal.iteba.ac.id/index.php/jurnalsiteba/index

[15] V. Pongsitammu, A. Renta Yani Simatupang, D. Annura, Y. Sari Dachi, and D. Rollando Harries, "Security of Modern Crypto Systems Based on Public Key Cryptography Algorithms," 2023. [Online]. Available: https://journal.iteba.ac.id/index.php/jurnalsiteba/indexSITEBA