A Novel System for Managing Encrypted Data Using Searchable Encryption Techniques

Vijay Govindarajan
Expedia Group, Seattle, USA

***Abstract*—The motivation for this study arises; from the insufficient security measures provided by cloud service providers, particularly with regard to data integrity and confidentiality. In today's digital landscape, nearly every international organization stores data in the cloud, whether through in-house servers or third-party providers. While encrypting data prior to storage addresses certain security concerns, it does not fully resolve the issue. Specifically, how can a server effectively process or search the data without decrypting it? This challenge is addressed by the concept of searchable encryption. Therefore, the objective of this study is to implement and evaluate a contemporary set of searchable encryption algorithms within a web-based platform. The study includes a comprehensive performance analysis of the implemented algorithms and an evaluation of the system based on the statistical outcomes of these algorithms. Therefore, this study aims to contribute to the advancement of secure and efficient methods for managing encrypted data in cloud environments. This study evaluates an image search system using the FAST protocol, achieving an average search time of 28.696 ms per image and an average deletion time of 0.557 seconds. While slower than FAST's benchmarks due to limited computational resources and additional processing steps, the system demonstrated reliable performance within its constraints. These results highlight the trade-offs between security, functionality, and performance, offering valuable insights for future optimizations in resource-constrained environments.**

***Keywords*—Cloud service providers; encrypting; security; web-based platform**

I. Introduction

The exponential growth of data generation has necessitated organizations to continuously expand their data storage infrastructure. The advent of cloud computing has provided a cost-effective and time-efficient alternative, enabling companies to scale their storage capabilities by outsourcing data to cloud-based platforms. It is estimated that 94% of enterprises currently utilize cloud services, and by 2025, over 100 zettabytes (1 trillion gigabytes) of data will be stored in the cloud. This underscores the increasing reliance of organizations on cloud storage solutions. Therefore, searchable encryption is a cryptographic approach designed to facilitate the storage and retrieval of encrypted data [1], [2], [3], [4], [5], [6]. This technology holds significant potential for organizations by enabling them to search and update encrypted data securely and efficiently. Despite the convenience offered by cloud storage, it is accompanied by notable security challenges, particularly concerning data privacy. When using third-party cloud providers; there is a risk of the provider accessing, controlling, or monitoring the stored data, as well as intercepting communications between the user and the server. To address these concerns, many organizations opt to encrypt their data before outsourcing it to the cloud. However, while encryption ensures data security, it complicates the process of efficient data retrieval. Communication during the retrieval process may expose sensitive information to the server, thereby undermining data privacy. However, searchable encryption seeks to address these challenges by allowing users to store, delete, and search encrypted data while maintaining its confidentiality from the server.

Therefore, this study focuses on implementing the Forward Private Searchable Symmetric Encryption (FAST) scheme within a web-based application. The application will interact with a cloud-based storage server, offering users three core functionalities i.e. uploading images associated with keywords to be stored in the cloud, deleting images from the cloud database using keywords, and searching for stored images in the cloud using keywords. The FAST scheme employs keyword-based protocols for indexing and retrieving data, ensuring secure and efficient data management. The user interface of the application will be designed to provide a seamless and intuitive experience, clearly distinguishing between the available options. Once the application is fully functional, its performance will be rigorously evaluated against the benchmark data presented in the [7].

The rapid adoption of cloud computing has revolutionized how organizations store, manage, and retrieve data. However, this shift has also introduced significant security and privacy challenges, particularly when sensitive data is stored on third-party cloud servers. While several solutions have been proposed to address these challenges, they often fall short in key areas, leaving critical vulnerabilities unaddressed.

Shortcomings of existing solutions are inadequate encryption methods, lack of forward privacy, performance bottlenecks and Focus on data encryption, not index encryption.

As noted in study [8], 90% of global enterprises use cloud computing, making it a critical component of modern IT infrastructure. Ensuring the security and privacy of data stored in the cloud is essential to maintaining user trust and compliance with regulations.

The study is as follows; the background will be provided in Section II. The relevant works are listed in Section III. The pre-implementation is covered in Section IV. The post-implementation is shown in Section V. The experimental analysis is carried out in Section VI, Discussion is given in

Section VII. Finally the paper is concluded in Section VIII with declarations at the end.

II. BACKGROUND

Before designing the system, it was crucial to develop a comprehensive understanding of the FAST protocols to ensure their effective implementation. The first protocol in the FAST framework is the setup protocol. This protocol is executed only once during the program's lifecycle. On the client side, the process begins by generating a master key which is a randomly generated binary string of lambda length. In this context, lambda represents the security parameter of the system, determining the level of encryption and security. Additionally, the client side initializes an empty map, denoted as sigma, which is designed to store metadata about the keywords associated with the database entries. On the server side, a corresponding empty map, denoted as *tau* (T) is initialized. This map is used to store server-side information about the documents in the database. Together, these maps establish the foundational structure for enabling secure keyword searches and updates in the system. By separating the roles of the client and server, the setup protocol ensures that sensitive operations, such as key generation and keyword mapping are confined to the client side, thereby enhancing the security and privacy of the stored data. The second protocol in the system is the update protocol, which manages all update operations, including adding and deleting documents from the database. It is important to note that each document addition or deletion requires an update operation. The parameters for this function include the master key, the client-side sigma map, the encrypted index of the document, the keyword associated with the document, the operation type (add or delete), and the server-side *tau* map. The protocol begins by generating a tag, which is the output of a pseudorandom function that uses the master key and the hash of the keyword as inputs. The keyword is then used as a key to retrieve data from the sigma map. If no data exists for this keyword, a new blank state is created, and the keyword's counter is set to zero. A new key is generated, and a new state is calculated using a pseudorandom permutation function. This function uses the newly generated key and the current state as inputs. This process ensures the state evolution in FAST is secure. The updated state and incremented counter are stored in the sigma map at the keyword's key. Next, the system prepares the data to be sent to the server. A variable is generated which concatenates the encrypted index of the document, the operation type (add or delete), and the XOR[1] of the new key with the hash of the tag and the new state. This approach ensures the confidentiality of the data by concealing it behind an XOR operation with a deterministic hash. The third and final protocol in FAST is the search protocol. This protocol facilitates all search operations in the system, such as identifying images to delete from the database after a deletion update. On the client side, the search protocol begins by generating a tag for the keyword, similar to the update protocol. Since the function used for this is deterministic, the tag remains consistent for the same keyword. The protocol checks the sigma map using the keyword as the key. If no data exists, the search returns no results, as this indicates no documents with the specified keyword have been added to the database. If data is found, it retrieves the current state and counter for the keyword, which are then sent to the server.

[1]A logical operation known as XOR (exclusive OR) produces true only when the inputs vary; otherwise, it produces false.

III. RELATED WORKS

Gaining insight into the vulnerabilities of cloud systems is essential to designing a secure solution. Several academic sources were reviewed to identify these challenges and their implications such as the study in [8] provides a comprehensive analysis of the security challenges inherent in cloud computing. It categorizes the various uses of cloud computing and highlights the security issues specific to each application. Of particular relevance to this study is the "Infrastructure as a Service" (IaaS) model, commonly referred to as Cloud Infrastructure Services (CIS), which is extensively discussed in study [8]. This model is crucial for understanding the security implications of using cloud storage for fetching, updating, and querying data. The study in [8] notes that "90% of global enterprises use cloud computing as part of their industries", underscoring the widespread reliance on cloud services. This insight reinforces the study's motivation to provide a secure solution for data storage and retrieval in cloud environments, given the current limitations in ensuring high levels of security. [9] provides a detailed analysis of how cryptographic algorithms are employed by organizations to maintain data security when utilizing cloud storage. It addresses critical issues related to data confidentiality and integrity, particularly when relying on third-party services for data storage. A key contribution of the study [9] is the proposal of multi-level encryption, which combines both Data Encryption Standard (DES)[2] and RSA[3] algorithms to enhance security. The process involves encrypting data with DES initially, followed by a second layer of encryption using RSA. The decryption process reverses this order, decrypting the RSA-encrypted data first and then applying DES decryption to retrieve the original file. This dual-layered approach strengthens data protection and ensures secure storage and retrieval processes. Given that this study employs a cloud-based system for storing and retrieving data, [10] is highly relevant as it offers insights into conventional security practices. The use of multi-level encryption aligns conceptually with the FAST approach utilized in this study, which also incorporates encryption methods to ensure data security. However, while the study [11] focuses on encrypting the actual data, FAST emphasizes the encryption of file indexes. Despite this distinction, the insights provided on cryptographic algorithms contribute valuable ideas for the encryption techniques that could be adopted in this study. The study in [11] introduces two forward-private searchable encryption algorithms, FAST and FastIO[4], which address common deficiencies in earlier encryption schemes. It builds

[2]The symmetric encryption algorithm known as DES uses a 56-bit key and works in blocks to encrypt data.

[3]Based on modular arithmetic, RSA is an asymmetric encryption method that encrypts and decrypts data securely using two keys (public and private).

[4]By optimizing input/output operations in programming and utilizing buffers to reduce latency, FastIO makes it possible for competitive coding to handle vast amounts of data efficiently.

upon the Sophos algorithms[5] by addressing their limitations and proposing improvements. A performance analysis of these algorithms is presented, evaluating search and update times across various database sizes and numbers of matching documents. These results are compared to Sophos, demonstrating the improvements achieved by the FAST algorithms. The protocols outlined in the study [12] will be directly implemented in this study, allowing for performance comparisons between the system developed here and the results documented in the study [13]. The research reinforces the motivation for this study by demonstrating that searchable encryption is a viable and effective solution for secure cloud data storage. The study in [14] provides a comprehensive overview of how searchable encryption can address data privacy concerns in cloud computing. It explores various Searchable Symmetric Encryption[6] (SSE) schemes, detailing their definitions and methodologies. By compiling these schemes chronologically, the study in [15] illustrates the evolution of SSE and how it facilitates efficient communication with cloud servers for secure data retrieval. The research emphasizes the importance of data privacy when storing information in the cloud, aligning with concerns highlighted in other cited sources. The study in [16] description of the methodologies underlying searchable encryption is particularly insightful, offering valuable inspiration for the system developed in this study. Although this study utilizes the FAST algorithm, the broader concepts outlined in study [17] remain applicable [18], [19], [20], [21], [22], [23], [24], [25], [26].

The problem of securing cloud-based data storage and retrieval has been a persistent challenge due to several limitations in previously proposed solutions:

A. Inadequate Encryption Methods

Many earlier solutions relied on single-layer encryption (e.g., DES or RSA alone), which is vulnerable to advanced attacks. For example, the study [9] highlights that while multi-level encryption (combining DES and RSA) improves security, it still focuses on encrypting the actual data rather than the searchable indexes, leaving room for vulnerabilities in search operations.

B. Lack of Forward Privacy

Traditional searchable encryption schemes often fail to ensure forward privacy, meaning that adding new data to the system could reveal information about past searches.

C. Performance Issues

Earlier schemes, such as Sophos, suffered from inefficiencies in search and update times, especially as the database size grew. The study in [11] demonstrates that FAST and FastIO significantly improve performance, but these solutions were not widely adopted or integrated into practical systems.

D. Focus on Data Encryption, Not Index Encryption

Many solutions, like those discussed in study [10], focus on encrypting the data itself but neglect the encryption of file indexes. This oversight can expose search patterns and metadata, compromising user privacy.

E. Our Contributions

The proposed approach in this study addresses these limitations by:

1) Emphasizing index encryption: Unlike previous solutions that focus on encrypting the actual data, this study prioritizes the encryption of file indexes using the FAST algorithm. This ensures that search patterns and metadata remain secure, even if the data itself is compromised.

2) Ensuring forward privacy: The FAST algorithm guarantees forward privacy, meaning that adding new data to the system does not reveal information about past searches. This is a significant improvement over earlier schemes.

3) Improving performance: By implementing FAST, the study achieves faster search and update times compared to traditional algorithms like Sophos, as demonstrated in study [11]. This makes the solution more practical for real-world applications.

4) Leveraging multi-level encryption concepts: While the study does not directly use DES and RSA, it incorporates the concept of multi-level encryption by encrypting both the data and the searchable indexes, ensuring comprehensive security.

IV. Pre-Implementations

This section outlines the fundamental structure of the system, detailing the components and their respective functionalities. The main page serves as the entry point for users and includes links to every page in the system, a brief explanation of the system's functionality, execution of FAST's setup protocol, and the setup protocol must be executed on this page, as it is the first step in initializing the system. Additionally, this page serves as a redirect after adding images to the database. The add image page is dedicated to providing a user interface for adding images to the database. Its features include a single image upload form with keyword input and a single image input, a multiple image upload form with keyword input, and multiple image input, validation to ensure uploaded files are images, and execution of FAST's update protocol. The page also offers two forms i.e. one for uploading multiple images under the same keyword and another for uploading a single image. Each image upload triggers a single update operation. The image search page is used to retrieve and display images from the database. It includes implementation of FAST's search protocol, a search form with a keyword input field, display of all matching images below the search form and metrics such as total time taken to retrieve images, number of matching images, average time taken to retrieve each image. These performance metrics are vital for evaluating the system's efficiency. The delete image page enables users to delete images from the database. It includes implementation of FAST's update protocol with the delete operation, a form to search for all images associated with a keyword, a confirmation form to delete the images, and

[5]To detect, stop, and lessen cyberthreats in real time, Sophos algorithms integrate behavior analysis, machine learning, and signature-based detection.

[6]Secure keyword searches over encrypted data are made possible by SSE, which maintains confidentiality while facilitating quick retrieval without the need for decryption.

confirmation of successful deletion after the operation. This page requires two forms to ensure that users confirm their intent to delete images. Images are only deleted upon submission of the second form.

A. System Design

This section provides a comprehensive overview of the development process for the system. The methodology employed was inspired by the Rapid Application Development (RAD) approach, specifically its rapid prototyping phase. In this approach, individual features are developed, tested, and refined iteratively until they are fully functional. Given the complexity of this study, where multiple components must seamlessly interact, extensive testing was conducted during development. Following the completion of development, whole-system testing was undertaken to ensure its reliability and functionality. The initial step of the study was configuring the development environment and creating the website's basic framework. This consisted of a single webpage devoid of links or content. Subsequently, the HyperText Markup Language (HTML)[7] and Cascading Style Sheets (CSS)[8] were adapted from the design specifications, resulting in the creation of the main page. Fig. 1 illustrates this outcome. As the main or introductory page, it contains only a brief overview of the website's purpose and links to pages for database manipulation. The subsequent task involved creating three additional pages and enabling functional navigation between them. This task proved more complex than anticipated due to the specific structural requirements of Django[9] projects. Typically, linking to another HTML file would involve referencing the file directly. However, Django utilizes a Python file, *urls.py*, to handle all routing between pages. This file employs name identifiers for each page, enabling consistent referencing throughout the study. After understanding this structure, navigation between pages was successfully implemented. Fig. 2 demonstrates how the *urls.py* file defines the URLs available on the website and assigns a unique name identifier to each page using the name parameter. This mechanism allows each page to be referenced in other templates. The middle parameter in the *urls.py* file specifies the function executed when a page is accessed or loaded. These functions, defined in the *views.py* file, are imported into the *urls.py* file by default. With this routing mechanism in place, creating a navigation bar for the main page became straightforward by consulting the Django documentation. In Django, to access variables or links stored on the server, the convention is to enclose the variable or link within braces and percentage signs (e.g., {% variable %}). Thus, instead of linking directly to an HTML file, links are directed to the URL paths defined in the *urls.py* file, with the page name enclosed in quotation marks. This structure facilitated the creation of templates for the additional pages.

Fig. 1. Main page of a website.

```
urlpatterns = [
    path("", views.home, name="home"),
    path("add-image", views.addImage, name="add-image"),
    path("image-search", views.imageSearch, name="image-search"),
    path("delete-image", views.deleteImage, name="delete-image")
]
```

Fig. 2. URLs.py.

1) Add image: The add image page plays a crucial role in the system, as it serves as the primary interface for storing images in the database. The page requires minimal input from the user i.e. a keyword to associate with the image and the image itself. The remaining fields necessary for storage are computed by the FAST protocols. At this stage of development, the page only includes a single image upload form. However, text below the navigation bar references multiple image uploads, as a future enhancement will introduce a multiple-image upload feature once FAST is fully integrated. This addition will significantly improve usability, as restricting uploads to a single image at a time would be inefficient for building a large image database. Fig. 3 illustrates the basic design of the add image page. Its corresponding HTML structure, shown in Fig. 4, forms the foundation for this functionality.

Fig. 3. Simple page for uploading images.

```
<form class = "form" name="SingleImageForm" id="SingleImageForm" enctype="multipart/form-data" method="POST" action="">
    <p class = "header"> Single Image Upload</p>
    {% csrf_token %}
    <label> Keyword: </label>
    <input type="text" required name="SingleUploadKeyword" id="SingleUploadKeyword">
    <br>
    <label> Image: </label>
    <input type="file" required name="SingleUploadImage" id ="SingleUploadImage" accept="image/*">
    <br>
    <input type="submit" value="Upload" name="SingleSubmit" id="SingleSubmit"/>
    <br>
</form>
```

Fig. 4. Simple HTML image upload form.

Further development will refine and expand upon this page to fully implement the intended features. The add image page is a fundamental component of the system, designed to facilitate the storage of images in the database. The page features a simple form with two input fields i.e. one for entering a keyword and the other for selecting an image file.

[7]Tags are used in HTML to describe elements such as text, images, links, and multimedia for browsers.

[8]In order to improve visual presentation, CSS creates and styles web content by regulating layout, colors, fonts, and responsiveness.

[9]Django is a high-level Python web framework enabling rapid development of secure, scalable web applications with reusable components and ORM.

The file input field is equipped with an accept attribute, which automatically filters for image files on the user's device. However, this does not completely restrict the upload to image files, as the user can modify the filter to display all files. To ensure the integrity of the system, server-side validation will be implemented later in the study to verify the uploaded file type. For security purposes, the inclusion of the {% csrf_token %} is vital. This token provides protection against Cross-Site Request Forgery[10] (CSRF) attacks, which could otherwise enable unauthorized actions to be executed by users. By including this token, the system ensures that submitted data remains unaltered and secure during the form submission process. Django not only supports this functionality but strongly encourages its implementation to enhance security. Additionally, the form's enctype attribute[11] is set to "multipart/form-data", which is necessary for handling file uploads securely. This encoding type ensures that all input data is encoded before being transmitted to the server, which is especially critical when handling files. In contrast, the alternative text/plain encoding type sends data as plaintext, which is insecure. Therefore, the use of "multipart/form-data" is essential for maintaining the system's security and reliability.

2) Image search page: The image search page provides functionality for retrieving stored images using keywords. Users are only required to input a keyword, as other necessary data is either pre-stored in the system or dynamically generated using the provided keyword (e.g., associated tags). The structure of this page closely resembles the add image form but excludes the file upload field, as it is unnecessary for this functionality. As with the previous page, a CSRF token is included to ensure the integrity of the keyword submitted to the server. This precaution protects the system against potential data tampering during the form submission process.

3) Image deletion page: The image deletion page is designed to enable users to remove images from the database. Unlike the process of adding images, deletion requires only a single input field i.e. the keyword. Upon submission, the system performs a search to identify any images associated with the provided keyword. If no matching images are found, no further actions are taken. Otherwise, the update protocol is invoked to remove the images from the database. Similar to the other pages, the form includes a CSRF token to ensure the security of the input data. This approach maintains consistency across all forms within the system and reinforces the overall security framework.

B. Setting Up and Connecting to the Database

Amazon Relational Database Service[12] (RDS) was selected for this study. AWS RDS is a widely used and well-documented platform, making it a popular choice for both individuals and organizations. Establishing a connection between the Django project and the database was more straightforward than initially anticipated. The *settings.py* file, automatically generated when creating a Django project, includes a dedicated section for defining database connections. By adding the required credentials (e.g., database name, user, password), as shown in Fig. 5, the study was successfully connected to the AWS database. The credentials used were obtained from the AWS Management Console[13] (AMC), except for the username and password, which were created during the initial server setup. Although the initial plan was to configure the database table using MySQL Workbench[14], Django's *models.py* file offers a more streamlined approach. By defining a class in *models.py*, a corresponding table is automatically created in the database. For this study, a class named ImageStorage was created with two fields i.e. index (a character field with a maximum length of 500, representing the encrypted index of the file. This length allows flexibility for future modifications), and ImageFile (a file field that specifies the file storage location using the upload_to parameter). A local media folder was designated for file storage to simplify debugging and ensure accessibility during development. The class also defines a function to return both the index and the image when displaying table contents in Django. This approach ensures that the stored data can be effectively verified and debugged.

Fig. 5. Database part of settings.py.

To verify the database connection, preliminary testing was conducted using Django's admin page. Fig. 6 demonstrates the admin page layout, where the newly created ImageStorage table is visible under the website tab. The next step involved adding data to the table through the admin interface, as shown in Fig. 7 and Fig. 8. Both fields—index and ImageFile—were successfully populated, confirming that the database connection and table configuration were functioning as intended. Ensuring a robust database connection is critical to the study's success, as all website pages rely on seamless communication with the database. Early testing was essential to identify and resolve potential issues, enabling efficient development and ensuring the system's overall performance and usability. Having established a working database connection, the next phase of development involves integrating communication between the website pages and the database.

C. TextConversion.js and Random.js Files

The *TextConversion.js* and *Random.js* files are integral components of the system, enabling data encoding, secure

[10]By deceiving users into performing unwanted actions on a trusted web application, CSRF takes advantage of user authentication.

[11]When submitting a form, particularly for file uploads, the `enctype` element in HTML indicates the encoding type for form data.

[12]A managed relational database service, Amazon RDS supports several database engines, scales, automates backups, and streamlines administration.

[13]Users can configure, monitor, and manage resources with the help of the AWS Management Console, a web interface for controlling AWS services.

[14] With its visual tools for MySQL, MySQL Workbench offers a single platform for database design, development, administration, and management.

index generation, and cryptographic key creation. These functions ensure compatibility between programming languages, maintain data integrity, and enhance system security. The primary purpose of the Text Conversion function, stored in *TextConversion.js*, is to convert strings into binary format by encoding each character. This approach was chosen because binary encoding preserves consistent meaning across different programming languages. Without this conversion, certain characters in a string, such as escape literals, could lead to discrepancies during data processing in Python. For instance, during the update protocol in FAST, the XOR operation may produce strings containing escape sequences such as */n*. In Python, this would be interpreted as a newline character, potentially corrupting the data and causing errors when the search protocol attempts to reconstruct the original string. The function, depicted in Fig. 9, processes an input string by iterating through each character, converting it into its binary representation, and appending it to a result string. To ensure the final result does not end with a trailing space, a conditional statement checks if the current character is the last index of the input string, and if so, it omits the addition of a space. This ensures a clean, correctly formatted binary string is returned for further use.

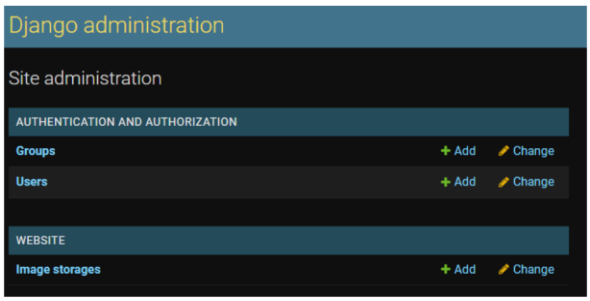


Fig. 6. Examine the admin page first.

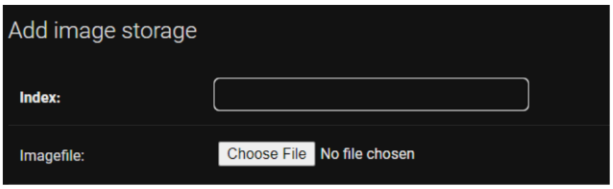


Fig. 7. Add image storage to the admin page.

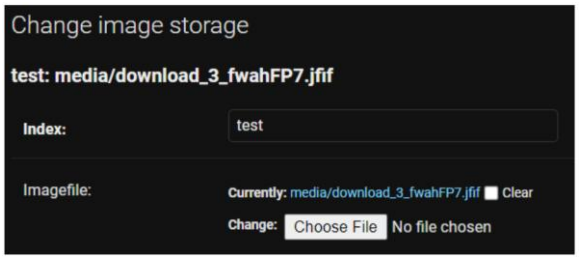


Fig. 8. First upload of the admin page.

```
function convertToBinary(string){
    result = "";
    for(var i = 0; i < string.length; i++){
        if(i == (string.length-1)){
            result += string[i].charCodeAt(0).toString(2)
        }
        else{
            result += string[i].charCodeAt(0).toString(2) + " ";
        }
    }
    return result;
}
```

Fig. 9. Text to binary function.

D. Generation Functions

1) Index generation function: The index generation function, illustrated in Fig. 10, is responsible for creating unique identifiers for data entries within the system. This function resides in *Random.js* and generates a random string with a length of 27 characters. The function begins by defining an empty string, tempInd, which will hold the generated index. A character set comprising all uppercase and lowercase letters (A-Z, a-z) and digits (0-9) is defined, resulting in 62 possible characters for each position in the index. This extensive character set significantly reduces the likelihood of generating duplicate indexes. A for loop is then employed to randomly select a character from the character set and append it to tempInd. This process is repeated until the string reaches the desired length of 27 characters. Once the loop completes, the fully constructed index is returned for use in the system. The use of a 62-character set and a length of 27 ensures that the probability of generating duplicate indexes within the scope of the study is extremely low, thereby enhancing the uniqueness and reliability of the identifiers.

```
function indGen(){
    var tempInd = "";
    var charSet = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
    var indLength = 27;
    for(var i = 0; i < indLength; i++){
        tempInd += charSet.charAt(Math.floor(Math.random()*charSet.length));
    }
    return tempInd;
}
```

Fig. 10. Index generation function.

2) Key generation function: The key generation function, shown in Fig. 11, is also located in *Random.js* and is designed to create cryptographic keys for use in the system. These keys adhere to the requirements of the FAST protocol, consisting exclusively of binary digits (0s and 1s) and having a length of 32 characters. The function begins by initializing an empty string to hold the generated key. A character set containing only 0 and 1 is defined to maintain consistency with the encryption and decryption protocols, particularly for Advanced Encryption Standard[15] (AES) operations. Similar to the index generation process, a for loop is used to randomly select characters from the binary character set. Each selected character is appended to the key string until it reaches the

[15]The symmetric encryption algorithm known as AES is frequently used to protect data with variable key lengths of 128 bits, 192 bits, and 256 bits.

specified length of 32 characters. The completed key is then returned to the calling function. This approach ensures that the generated keys are both secure and compatible with the cryptographic requirements of the system, enabling efficient encryption and decryption processes. These functions collectively contribute to the robustness and security of the system, ensuring data integrity, compatibility across programming languages, and adherence to cryptographic standards.

```
function keyGen(){
    var charSet = "01";
    var keyLength = 32;
    var key = "";
    for(var i = 0; i < keyLength; i++){
        key += charSet.charAt(Math.floor(Math.random()*charSet.length));
    }
    return key;
}
```

Fig. 11. Key generation function.

3) State generation function: The state generation function, illustrated in Fig. 12, is conceptually similar to the key generation function, with the primary difference being the length of the generated state. The state length is fixed at 16 characters to align with the block size requirements of AES ECB[16] encryption without padding. This choice ensures that the resulting ciphertext remains concise, allowing for efficient decryption and enabling the addition of more states to the system. With a 16-character length, the number of possible states that can be generated is 65,536. Notably, states themselves do not need to be unique for the system to function correctly. Instead, the uniqueness lies in the pairing of a keyword and its corresponding state, which ensures the proper operation of the FAST protocol. For each state, a binary number is generated only once, so even if multiple states in the system have the same starting value, FAST remains functional due to the unique keyword-state pairing.

```
function stateGen(){
    var charSet = "01";
    var stateLength = 16;
    var state = "";
    for(var i = 0; i < stateLength; i++){
        state += charSet.charAt(Math.floor(Math.random()*charSet.length));
    }
    return state;
}
```

Fig. 12. State generation function.

E. Implementing the FAST Setup Protocol

The implementation of the FAST setup protocol was a prerequisite for developing the site. This protocol involves generating a master key and a sigma map on the client-side and a *tau* map on the server-side. To avoid resetting the system with each new session, the master key, sigma map, and *tau* map are stored in plaintext text files for simplicity, alongside an indicator text file to track the session state. While this approach was chosen due to time constraints, it should be noted that plaintext storage is inherently insecure and should ideally be replaced with encrypted storage in future iterations. The setup process is initiated on the server-side. Upon detecting that the text files are empty, the server initializes the *tau* map as an empty dictionary ({}) and passes this context to the main page, signaling that setup is incomplete. If the files contain data, the server reads and loads the variables into memory and passes the sigma map and master key to the client-side. On the client-side, the setup script determines whether setup is complete by checking the indicator file. If setup is incomplete, the sigma map and master key are generated and stored in the browser's session storage, enabling their retrieval across other pages. If setup is already complete, the existing values are loaded and stored similarly. Debugging variables are used to ensure consistency across files.

Therefore, the add image page allows users to upload an image and associate it with a keyword. The process includes both client-side and server-side operations. On the client-side, the index generation function is triggered when the user clicks the form's submit button. A hidden input field is dynamically populated with the generated index before the form is submitted. This ensures that the index is included in the POST request to the server. On the server-side, the *views.py* function processes the form submission. It verifies whether the form submission is a data upload or a page render. If an image is uploaded, it validates the file type to prevent malicious scripts from being uploaded. Valid images are stored in the database, along with their respective index and keyword. Upon successful upload, the user is redirected to the main page with a confirmation message. Testing of this implementation, as demonstrated in Fig. 13, 14, and 15, confirmed that the system successfully stores a single image at the generated index.

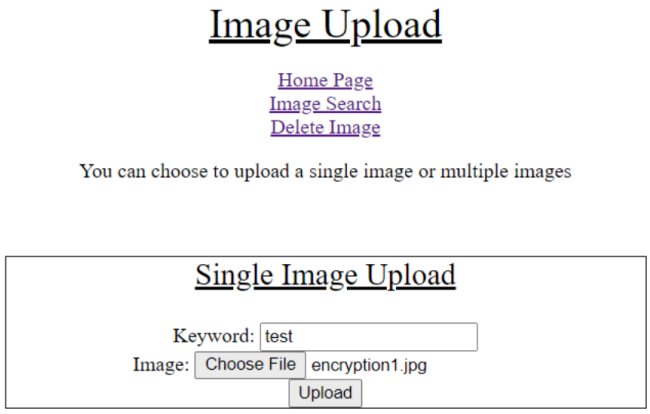


Fig. 13. First-stage test of the image upload page.

Fig. 14. The initial test of the image upload page was successful.

[16]The AES ECB (Electronic Codebook) mode is vulnerable to pattern leakage yet independently encrypts data in fixed-size blocks without chaining.

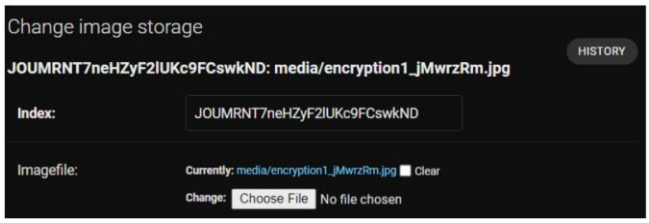


Fig. 15. View of the admin page during the initial image uploading test.

However, on the client-side, the event listener for the form was updated to call the addIndexUpdate function, which handles the upload process. This function begins by fetching the sigma map and master key from the session storage. To support encryption, the CryptoJS[17] library was incorporated for cryptographic operations. The FAST update protocol starts with the generation of a tag. The keyword is encoded in UTF-8 format, hashed using SHA-256[18], and encrypted using AES ECB, with the hash serving as the plaintext and the master key as the encryption key. The next steps involve generating the index and encryption key using pre-existing functions. If the keyword is already in the sigma map, the corresponding state and counter are retrieved. The state is encrypted using AES ECB, with the state encoded in Base64 and the key in UTF-8 format. The updated state and incremented counter are then stored back in the sigma map. If the keyword is not in the map, a new state and counter are generated, and the same encryption process is applied. The final step in the protocol involves generating the *u* and *e* variables. The *e* variable is created by XOR-ing the hash of the tag and state with the operation type, key, and index. This result is then converted to binary format. To facilitate the server-side processing, additional hidden input fields were added to the form for the *u*, *e*, index, sigma map, and master key. These fields are dynamically populated before submission.

V. Post-Implementations

On the server-side, the views.py function processes the uploaded data. The function first verifies the file type to ensure only valid image formats are accepted. Invalid files trigger an error message, redirecting the user to the main page. If the file is valid, the *u*, *e*, and updated sigma map are stored using custom functions. The T map is always updated to ensure synchronization. The session indicator is updated, and the master key is stored in its respective file to maintain consistency across sessions. The integration of the FAST update protocol ensures that the system is secure and efficient while adhering to cryptographic standards. Testing confirmed the successful implementation of all features for single image uploads with FAST integration. The process of testing the single image upload functionality is documented through the Fig. 16, 17, and 18. These figures illustrate that the system performs correctly on a fresh setup, with a single image being uploaded successfully to the database and reflected in the sigma map. Following this, tests were conducted to check for stacking encrypted states and new fresh states, ensuring that the system handles multiple image uploads effectively. Fig. 19 and 20 document this process, where a second image with the same keyword and a completely new keyword were uploaded successfully, demonstrating the functionality of the single image upload form.

To expand the image upload feature, the system was enhanced to support the uploading of multiple images simultaneously. This involved creating a new form, as detailed in Fig. 21, which allows for the selection of multiple files. The key modification to this form was the introduction of the "multiple" parameters in the file input, enabling multiple file selections. This form is rendered just below the single image upload form, as shown in Fig. 22.

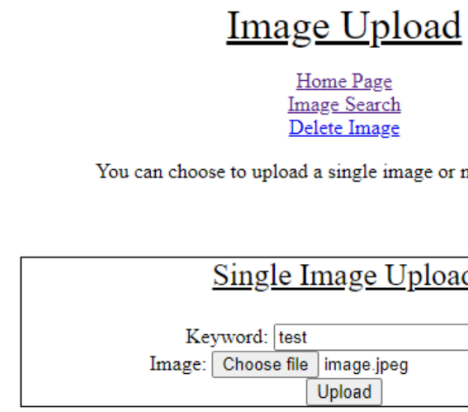


Fig. 16. Final try of uploading a single image.

Fig. 17. After the last attempt of uploading a single image, redirect.

Fig. 18. Console output for the test of uploading a single image.

Fig. 19. Single image upload test in an evolving state.

Fig. 20. Console output for the test of the final image upload.

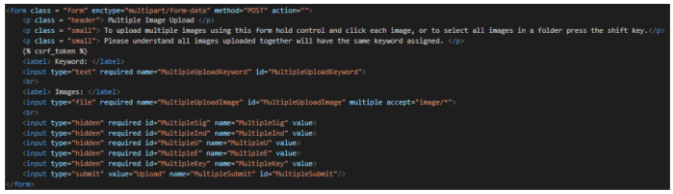


Fig. 21. Uploading multiple images from HTML.

[17]A JavaScript package called CryptoJS offers cryptographic methods for safe data encryption and hashing, including AES, SHA, and HMAC.

[18]A popular cryptographic hash algorithm for data security and integrity applications, SHA-256 generates a 256-bit result.

Fig. 22. A form for uploading multiple images was created.

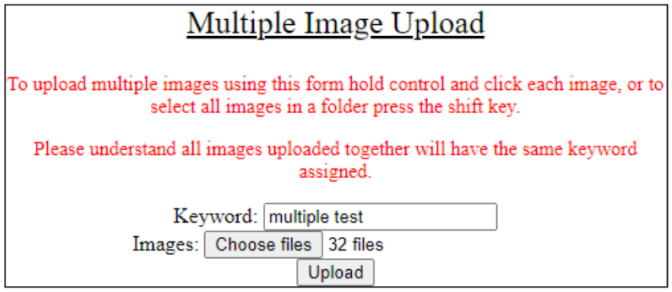


Fig. 23. HTML test for multiple upload forms.

To handle the multiple image uploads, a new event listener was implemented, which triggers a function named "multipleUpload". This function is responsible for processing the multiple files, where dictionaries are used to store the indices, *u*, and *e* variables from each file's update process. The update function was adapted to handle multiple uploads by iterating through each file, generating necessary variables, and sending them to the server. In the server-side processing, the images are validated to ensure that only acceptable file types are uploaded. If any invalid files are detected, the upload process is aborted. Once validated, the images are uploaded to the database, and the total time taken for the process is calculated. If any invalid file types are uploaded, an error message is displayed. This multi-image upload feature was thoroughly tested, and the system behaved as expected, accepting legitimate image files and rejecting non-image files, as shown in Fig. 23–26.

Fig. 24. Successful upload of the main page.

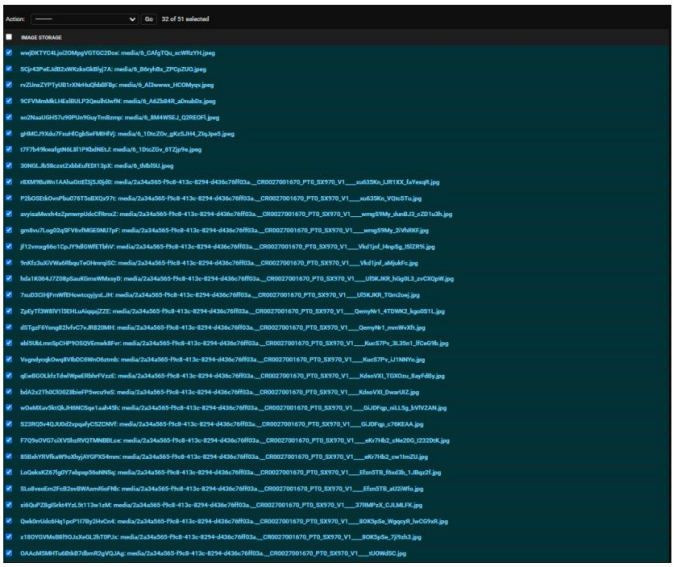


Fig. 25. The admin page displaying the images that were successfully uploaded.

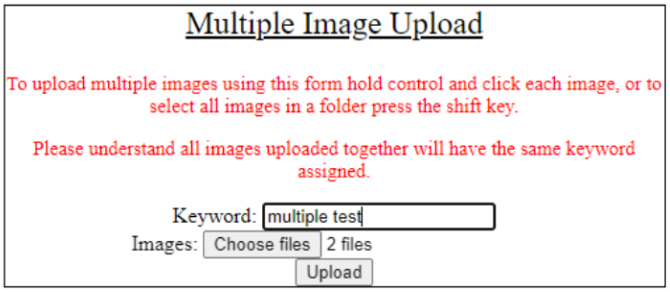


Fig. 26. Two text files are attempted to be uploaded.

The search functionality was developed using FAST to search for images in the database. The process began by retrieving the sigma map and master key from session storage. The search function was designed to send both the tag state and counter, along with an indicator specifying whether the keyword was already present in the sigma map. The server-side processing for the search involves retrieving the keyword, state, and counter from the POST request and performing a search through the sigma map. The search iterates through the states to identify whether the images exist and regenerates the previous states using the *u* and *e* variables. If no matches are found, a message is displayed indicating the absence of results. The system was tested using the "multiple test" keyword, successfully retrieving and displaying images, as seen in Fig. 27 and 28.

Fig. 27. Enter the search test keyword.

Fig. 28. Keyword search results.

The image deletion functionality was developed similarly to the image upload form but with a focus on removing images from the database. The process involved creating a search function to identify images associated with a keyword and then using a second form for deletion. The deletion form was triggered after a search, with the user selecting images for deletion. The image deletion process was handled by the "delIndexUpdate" function, which works similarly to the addIndexUpdate function but with an operation type set to "del". The server-side processing for deleting images involved iterating through the selected images, updating the corresponding data in storage, and deleting the images from the database. Upon completion, the system calculates and displays the time taken to delete the images. The deletion page was tested successfully, as shown in Fig. 29, 30, and 31, with the system correctly identifying and deleting the selected images from the database.

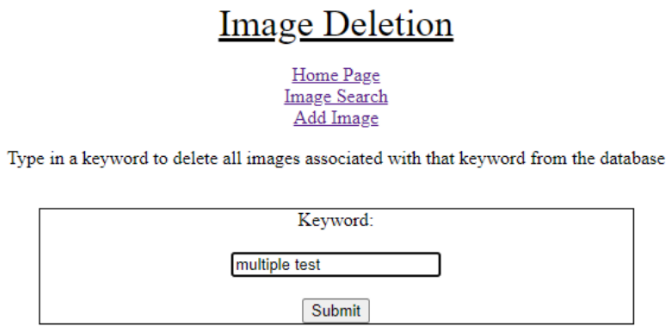


Fig. 29. Remove the image from the test.

Fig. 30. Remove the second form test image.

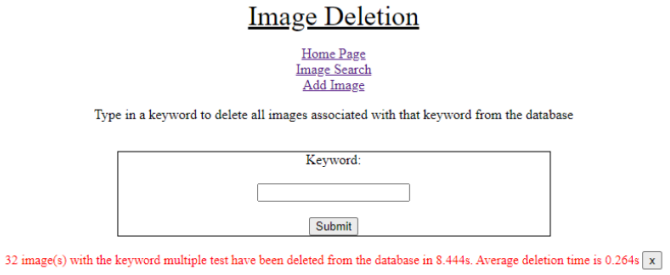


Fig. 31. Images were successfully removed.

VI. Experimental Results

This section outlines the performance results obtained from multiple runs of the program. To ensure consistency and reliability, each testing iteration began with a fresh setup, wiping the project clean. Images were added to the system using the "add image" page, keywords were searched using the "search image" page, and all images were subsequently deleted using the "delete image" page. These results were documented to reflect the implementation of all FAST protocols, along with the additional processing required by the system. It is expected that the performance results will be slower than the raw performance of FAST due to the additional overhead introduced by system-specific operations. It is anticipated that the average time taken per image would increase as the state lengthens. This is because longer states require more processing power to generate ciphertext, with the base64 ciphertext increasing in size with each iteration. For thoroughness, different keywords were alternated to store the images, and results were recorded in chronological order. Initial testing was conducted on a desktop computer with a quad-core i5 processor (3.4GHz) and 8GB of RAM. To provide a comparative analysis, the same project was tested on a laptop with a dual-core i3 processor (2.5GHz) and 16GB of RAM. The objective was to evaluate the impact of CPU speed and available memory on performance, particularly when handling larger datasets. The performance comparison between the two systems yielded expected results. Despite the laptop having double the RAM, the desktop's superior CPU speed compensated for the reduced memory. Both systems performed similarly when processing a comparable total number of images, with the desktop generally showing faster update times due to its stronger processor.

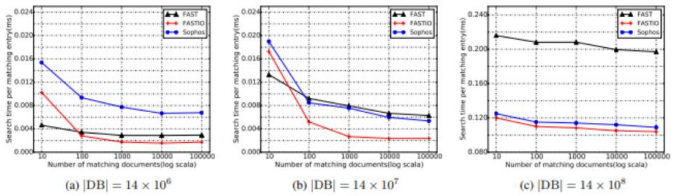
The average update times for the FAST update protocol are illustrated in Fig. 32. While these results indicate significantly slower performance compared to FAST's raw update times, this discrepancy can be attributed to the factor that the system's custom functions involve additional data handling, requiring adjustments to variables as they transition from client-side to server-side. This adds processing overhead to the system. Updating files necessitates clearing existing

data before rewriting, which further impacts performance. The results demonstrate that while the system's performance is slower than FAST's raw update times, this is an expected outcome given the added complexity of data manipulation and file handling. The comparative analysis between the desktop and laptop systems highlights the significant role of CPU speed in managing larger datasets, even when memory capacity differs. This provides valuable insights into the trade-offs between processing power and memory allocation in system performance.

		FAST	FASTIO	Sophos
Local	Throughput (ops/s)	54060	76100	4890
	Single update time (ms)	0.018	0.013	0.20
WAN	Throughput (ops/s)	21650	31080	2990
	Single update time (ms)	0.046	0.032	0.334

Fig. 32. FAST update time.

However, the image search functionality was tested by uploading a total of 2,877 images to the database and associating them with five distinct keywords. For each keyword, multiple searches were conducted, and the average search time was calculated. The overall average search time for a single image across all keywords was determined to be 28.696 ms. The search results obtained from the system were compared against the performance metrics outlined in [7]. Fig. 33 provides a reference to FAST's performance graph. Given the relatively small size of the database used in this study—due to limited computational resources—the comparison was made to the leftmost section of the graph, which represents the smallest database size used in FAST's evaluation (albeit still significantly larger than the database used here).



(a) $|DB| = 14 \times 10^6$ (b) $|DB| = 14 \times 10^7$ (c) $|DB| = 14 \times 10^8$

Fig. 33. Statistics on FAST search performance.

VII. Discussion

The results from my testing were significantly slower than the search times reported in [7]. This discrepancy can be attributed to several factors such as the computational power of the systems used for testing was limited, which likely contributed to the slower search times. The system performs extra processing during the search phase, including decoding binary strings and applying XOR operations to each character. This increases the computational workload significantly. To facilitate passing the XORed string to Python, a binary encoding file was introduced. Without this, Python would occasionally generate escape literals that caused the system to malfunction during debugging. Although the search protocol in this system exhibits slower performance compared to FAST's benchmarks, this is a reasonable outcome given the additional processing requirements and limited hardware capabilities. Furthermore, as the database size increases, the performance degradation becomes more pronounced. Nevertheless, the system demonstrates functional reliability and acceptable performance within the constraints of the testing environment. The results for image deletion were compared to the performance benchmarks in FAST's update protocol, as presented in Fig. 32. The average deletion time of 0.557 seconds per image is nearly 10,000 times slower than the single update time achieved in a WAN setting. This significant disparity is largely attributed to the additional processing requirements during the deletion process. For instance, accessing certain variables—such as the index—can only be achieved during this stage, which adds to the overall time. It is important to note that this time does not include the GET form on the deletion page. If included, its processing speed would align closely with the search page's results.

VIII. Conclusion and Future Works

Originally, the plan was to acquire the database, format it using MySQL Workbench or another MySQL software, and then begin storing data. However, this approach evolved once we became more familiar with Python's Django library. The library's functionality allowed for the creation of tables directly within the project using the Models.py file, which streamlined the process and made the database structure highly adaptable during the project's development. While the database functioned effectively and met its intended purpose, its performance was constrained by the computational limitations of the system. Specifically, the use of two slow virtual CPUs resulted in slower data storage times. Nonetheless, the database served as a reliable foundation for the project. The main page was designed to execute the setup protocol from FAST whenever a user accessed it. The implementation was expanded to save variables generated during the setup protocol into files, enabling sessions to be resumed later. This page was developed successfully, with the navigation bar providing seamless interaction with other pages. By organizing the protocols across separate pages, debugging was simplified. While the main page's development followed the original plan, it took approximately one week to complete due to initial challenges in understanding Django's file structure requirements. Specific directories and files needed precise organization to ensure functionality. This page required the most development time, taking approximately 1.5 weeks to complete. Initially, creating both a single-upload form and a multiple-upload form posed challenges in handling POST requests since both forms relied on the same HTTP method. Differentiating the requests required identifying the specific button used to send the request. Much of the time was spent debugging the communication between different pages rather than directly handling image uploads. While this page caused delays, its completion was a critical milestone in the project. The image search page was relatively straightforward to implement. The primary challenge was on the server side, specifically with regenerating data stored in the variable *e*. To address this, a binary encoding system was introduced, preventing character corruption across programming languages. The deletion page was comparatively easier to develop, as it leveraged the code from the add and search pages. By combining and adapting the algorithms from these pages, the final result allowed for deleting all images associated with a specific keyword from the database.

To tackle the computational complexity of advanced cryptographic schemes like FAST, future research should focus on hardware acceleration (e.g., GPUs, TPUs), algorithmic optimizations (e.g., efficient data structures, parallel processing), and lightweight cryptographic primitives to reduce processing time. Exploring distributed and decentralized architectures, such as blockchain or edge computing, can improve scalability and resource utilization. Additionally, integrating machine learning for predictive caching and adopting quantum-resistant algorithms will ensure the system remains efficient and secure in the long term.

IX. Declarations

A. Funding

No funds, grants, or other support was received.

B. Conflict of Interest

The authors declare that they have no known competing for financial interests or personal relationships that could have appeared to influence the work reported in this paper.

C. Data Availability

Data will be made on reasonable request.

D. Code Availability

Code will be made on reasonable request.

References

[1] G. S. Kashyap et al., "Revolutionizing Agriculture: A Comprehensive Review of Artificial Intelligence Techniques in Farming," Feb. 2024, doi: 10.21203/RS.3.RS-3984385/V1.

[2] S. Wazir, G. S. Kashyap, and P. Saxena, "MLOps: A Review," Aug. 2023, Accessed: Sep. 16, 2023. [Online]. Available: https://arxiv.org/abs/2308.10908v1

[3] H. Habib, G. S. Kashyap, N. Tabassum, and T. Nafis, "Stock Price Prediction Using Artificial Intelligence Based on LSTM– Deep Learning Model," in Artificial Intelligence & Blockchain in Cyber Physical Systems: Technologies & Applications, CRC Press, 2023, pp. 93–99. doi: 10.1201/9781003190301-6.

[4] G. S. Kashyap, K. Malik, S. Wazir, and R. Khan, "Using Machine Learning to Quantify the Multimedia Risk Due to Fuzzing," Multimed. Tools Appl., vol. 81, no. 25, pp. 36685–36698, Oct. 2022, doi: 10.1007/s11042-021-11558-9.

[5] G. S. Kashyap et al., "Detection of a facemask in real-time using deep learning methods: Prevention of Covid 19," Jan. 2024, Accessed: Feb. 04, 2024. [Online]. Available: https://arxiv.org/abs/2401.15675v1

[6] F. Alharbi and G. S. Kashyap, "Empowering Network Security through Advanced Analysis of Malware Samples: Leveraging System Metrics and Network Log Data for Informed Decision-Making," Int. J. Networked Distrib. Comput., pp. 1–15, Jun. 2024, doi: 10.1007/s44227-024-00032-1.

[7] X. Song, C. Dong, D. Yuan, Q. Xu, and M. Zhao, "Forward Private Searchable Symmetric Encryption with Optimized I/O Efficiency," IEEE Trans. Dependable Secur. Comput., vol. 17, no. 5, pp. 912–927, Sep. 2020, doi: 10.1109/TDSC.2018.2822294.

[8] M. B. Yassein, S. Aljawarneh, E. Qawasmeh, W. Mardini, and Y. Khamayseh, "Comprehensive study of symmetric key and asymmetric key encryption algorithms," in Proceedings of 2017 International Conference on Engineering and Technology, ICET 2017, Institute of Electrical and Electronics Engineers Inc., Jul. 2017, pp. 1–7. doi: 10.1109/ICEngTechnol.2017.8308215.

[9] Z. Khanam and M. N. Ahsan, "Implementation of the pHash algorithm for face recognition in a secured remote online examination system," Int. J. Adv. Sci. Res. Eng., vol. 4, no. 11, pp. 01–05, 2018, doi: 10.31695/ijasre.2018.32917.

[10] X. Li, T. Lai, S. Wang, Q. Chen, C. Yang, and R. Chen, "Weighted feature pyramid networks for object detection," in Proceedings - 2019 IEEE Intl Conf on Parallel and Distributed Processing with Applications, Big Data and Cloud Computing, Sustainable Computing and Communications, Social Computing and Networking, ISPA/BDCloud/SustainCom/SocialCom 2019, Dec. 2019, pp. 1500–1504. doi: 10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00217.

[11] M. Chase and S. Kamara, "Structured encryption and controlled disclosure," in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Springer Verlag, 2010, pp. 577–594. doi: 10.1007/978-3-642-17373-8_33.

[12] V. M. Vilić, "Dark web, cyber terrorism and cyber warfare: Dark side of the cyberspace," Balk. Soc. Sci. Rev., vol. 10, no. 10, pp. 7–24, 2017.

[13] K. Pavani and P. Sriramya, "Enhancing public key cryptography using RSA, RSA-CRT and N-Prime RSA with multiple keys," in Proceedings of the 3rd International Conference on Intelligent Communication Technologies and Virtual Mobile Networks, ICICV 2021, Institute of Electrical and Electronics Engineers Inc., Feb. 2021, pp. 661–667. doi: 10.1109/ICICV50876.2021.9388621.

[14] L. Gong, K. Qiu, C. Deng, and N. Zhou, "An image compression and encryption algorithm based on chaotic system and compressive sensing," Opt. Laser Technol., vol. 115, pp. 257–267, Jul. 2019, doi: 10.1016/j.optlastec.2019.01.039.

[15] M. Prerna, A. Sachdeva, and P. Mahajan, "A Study of Encryption Algorithms AES, DES and RSA for Security AStudyofEncryptionAlgorithmsAESDESandRSAforSecurity A Study of Encryption Algorithms AES, DES and RSA for Security," Type Double Blind Peer Rev. Int. Res. J. Publ. Glob. Journals Inc, vol. 13, 2013.

[16] D. Awasthi and V. K. Srivastava, "Hessenberg Decomposition-Based Medical Image Watermarking with Its Performance Comparison by Particle Swarm and JAYA Optimization Algorithms for Different Wavelets and Its Authentication Using AES," Circuits, Syst. Signal Process., pp. 1–32, Mar. 2023, doi: 10.1007/s00034-023-02344-z.

[17] A. Nadeem and M. Y. Javed, "A performance comparison of data encryption algorithms," in Proceedings of 1st International Conference on Information and Communication Technology, ICICT 2005, 2005, pp. 84–89. doi: 10.1109/ICICT.2005.1598556.

[18] G. S. Kashyap, D. Mahajan, O. C. Phukan, A. Kumar, A. E. I. Brownlee, and J. Gao, "From Simulations to Reality: Enhancing Multi-Robot Exploration for Urban Search and Rescue," Nov. 2023, Accessed: Dec. 03, 2023. [Online]. Available: https://arxiv.org/abs/2311.16958v1

[19] P. Kaur, G. S. Kashyap, A. Kumar, M. T. Nafis, S. Kumar, and V. Shokeen, "From Text to Transformation: A Comprehensive Review of Large Language Models' Versatility," Feb. 2024, Accessed: Mar. 21, 2024. [Online]. Available: https://arxiv.org/abs/2402.16142v1

[20] G. S. Kashyap, A. Siddiqui, R. Siddiqui, K. Malik, S. Wazir, and A. E. I. Brownlee, "Prediction of Suicidal Risk Using Machine Learning Models," Dec. 25, 2021. Accessed: Feb. 04, 2024. [Online]. Available: https://papers.ssrn.com/abstract=4709789

[21] F. Alharbi, G. S. Kashyap, and B. A. Allehyani, "Automated Ruleset Generation for 'HTTPS Everywhere': Challenges, Implementation, and Insights," Int. J. Inf. Secur. Priv., vol. 18, no. 1, pp. 1–14, Jan. 2024, doi: 10.4018/IJISP.347330.

[22] G. S. Kashyap, A. E. I. Brownlee, O. C. Phukan, K. Malik, and S. Wazir, "Roulette-Wheel Selection-Based PSO Algorithm for Solving the Vehicle Routing Problem with Time Windows," Jun. 2023, Accessed: Jul. 04, 2023. [Online]. Available: https://arxiv.org/abs/2306.02308v1

[23] M. Kanojia, P. Kamani, G. S. Kashyap, S. Naz, S. Wazir, and A. Chauhan, "Alternative Agriculture Land-Use Transformation Pathways by Partial-Equilibrium Agricultural Sector Model: A Mathematical Approach," Aug. 2023, Accessed: Sep. 16, 2023. [Online]. Available: https://arxiv.org/abs/2308.11632v1

[24] N. Marwah, V. K. Singh, G. S. Kashyap, and S. Wazir, "An analysis of the robustness of UAV agriculture field coverage using multi-agent reinforcement learning," Int. J. Inf. Technol., vol. 15, no. 4, pp. 2317–2327, May 2023, doi: 10.1007/s41870-023-01264-0.

[25] S. Wazir, G. S. Kashyap, K. Malik, and A. E. I. Brownlee, "Predicting the Infection Level of COVID-19 Virus Using Normal Distribution-Based Approximation Model and PSO," Springer, Cham, 2023, pp. 75–91. doi: 10.1007/978-3-031-33183-1_5.

[26] S. Naz and G. S. Kashyap, "Enhancing the predictive capability of a mathematical model for pseudomonas aeruginosa through artificial neural networks," Int. J. Inf. Technol. 2024, pp. 1–10, Feb. 2024, doi: 10.1007/S41870-023-01721-W.