Securing Embedded Metadata with Symmetric and Asymmetric Encryption

MSc Research Project
Cyber Security

Dmitry Cherniy
Student ID: 19230265

School of Computing
National College of Ireland

Supervisor: Ross Spelman

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Dmitry Cherniy……………………………………………………………………………………
Student ID:	19230265…………………………………………………………………………………………
Programme:	Cybersecurity…………………………………………………… **Year:** 2021……………
Module:	MSc Internship……………………………………………………………………………………
Supervisor:	Ross Spelman ……………………………………………………………………………………
Submission Due Date:	16/08/2021…………………………………………………………………………………………
Project Title:	Securing Embedded Metadata with Symmetric and Asymmetric Encryption …………………………………………………………………………………….
Word Count:	8152…………………………… **Page Count:** 26…………………………………….

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: ………………………………………………………………………………………………………

Date: 16/08/2021………………………………………………………………………………………………

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Securing Embedded Metadata with Symmetric and Asymmetric Encryption

Dmitry Cherniy
Student 19230265

Abstract

The penetration of information technologies into our lives has led to fundamental changes in our work, study, business, and leisure. It allows us to think, plan and make decisions in new ways. However, accelerating technological changes have created challenges related to digital security and user privacy. In recent years, unsecured metadata has led to serious security breaches, exposing businesses, individuals, and organisations to financial and reputational losses. Such security issues have become a significant concern and demonstrate the need to protect vital information about the original data. This report addresses common security issues related to embedded metadata. The report proposes a scheme for protecting embedded metadata using symmetric and asymmetric cryptography algorithms and describes the proposed model in detail.

Keywords - Metadata Encryption, XXTEA, AES, RSA, ElGamal, ECC, ECDH, Digital Signature, DSA, ECDSA, SHA-256

1 Introduction

In 2020 more than 4 billion people were using the Internet, and the audience of social networks has exceeded 3.8 billion. Businesses and organisations continue to expand their online activities, and the volume of digital transactions is constantly growing. However, despite all the benefits, the Internet poses privacy and security concerns for its users. Storing and delivering digital information creates many challenges. Among the most common problems for individuals and businesses is the privacy and security of digital data. Every day, users send billions of files over the Internet, which may contain digital traces. The information embedded in digital files is called metadata. It provides additional information about the files, specifications and version of the software used to create the file, date and time of creation, author name, geolocation, etc. Disclosure of this information can create privacy issues for a business, individual or organisation. Recent events, in particular, the ongoing Snowden affair, revealed widespread global surveillance by the U.S. government since 2013[1]. According to the NSA and the CIA former officials, governments often rely solely on metadata to process and identify the information collected during mass surveillance operations. Although the information contained in metadata seems trivial, yet it has the potential to create serious issues.

[1] https://www.bbc.com/news/world-us-canada-23123964

The embarrassment faced by the U.K. government in 2003 is a typical example of such a scenario. In 2003 the U.K. government presented a report to U.N. about the situation in Iraq. But report metadata showed that most of this 19-page dossier is a rewriting based primarily on open publications by three Iraqi researchers (Al-Marashi, 2006). This raised serious suspicions about the report's quality, reliability, and originality, resulting in national embarrassment. During the war in Iraq in 2007, insurgents pinpointed the location of the American Apache helicopters by examining the metadata of photographs posted on the Internet. As a result, the enemy carried out a mortar attack on the compound, destroying four AH-64 helicopters[2]. In another example, American law firm Venable won the case for its client by studying the defendant's document's system metadata. It was revealed that the last file save occurred earlier than the previous print, which experts on both sides acknowledged could not have been the case. The court found this evidence sufficient and awarded the plaintiff $ 20 million in compensation plus legal costs[3].

Metadata is essential to facilitate systems interoperability and ensures the immutability of the structure of an electronic document during processing operations. It is also used to set licensing restrictions on the distribution of information, indicating the author of the content. At the same time, metadata embedded in documents can leak confidential information. Therefore, it is crucial to protect metadata and restrict its access only to authorised users.

Previous research has focused chiefly on protecting embedded metadata using symmetric encryption methods (Wijayanto et al., 2016; Bhangale, 2019), hierarchical or group-based models for metadata access control (Lepsoy, 2015) and digital watermarking techniques through the digital watermarking algorithms (Faiz bin Jeffry and Mammi, 2017). The model proposed in this research paper provides metadata security by using symmetric and asymmetric encryption algorithms. The proposed design achieves metadata security, integrity, authentication and users privacy without revealing sensitive information. Furthermore, the proposed scheme is efficient in terms of encryption speed and computational operations compared to the state-of-the-art methods described in the literature. This paper seeks to address the following research question: "*How can symmetric and asymmetric encryption combined ensure the security, integrity, and confidentiality of Embedded Metadata*?"

2 Related Work

2.1 Previous approaches for Securing Embedded Metadata

Metadata is data about data that refers to additional information about the data object. It provides information about the characteristics and properties that describe entities, allowing automatically search and manage them in large information flows. Metadata is an essential component of any document, music file, video recording, or image file. It is assigned automatically and includes a description of the file, titles and subheadings, author and editor, date and time of creation, geolocation, version and technical characteristics of the software, etc. Unsecure metadata is a rapidly increasing threat to digital security largely ignored because metadata is typically hidden from users. Sometimes embedded metadata can be more valuable

[2] https://www.defensetech.org/2012/03/15/insurgents-used-cell-phone-geotags-to-destroy-ah-64s-in-iraq
[3] https://www.venable.com/insights/publications/2015/10/venables-20millionplussanctions-trade-secrets-win

than the file itself. It can serve as a source of information about a potential victim as part of the social engineer's attack. Searching for metadata in image files is one of the stages of the "doxing" technique, which has already become a prevalent practice of collecting information about a person of interest on the Internet for various purposes[4]. There have been numerous studies on securing metadata, and many approaches have been proposed. Mamta et al. (2020) suggested encode metadata in the image file using a secure data strip. Encoded metadata is embedded in a data strip and added to the image. However, lossy compression algorithms (JPEG, PNG, etc.) can damage the strip, resulting in the loss of metadata. Faiz bin Jeffry and Mammi, (2017) proposed digital watermarking techniques to add EXIF information into image files through the digital watermarking algorithms. The drawback of this method is similar to the previous approach, image cropping and compression methods used by processing algorithms can damage the watermark containing metadata. Bane and Minnear, (2017) proposed a system for prioritising, filtering and normalising metadata from files. The metadata associated with the files is saved and stored separately in the storage repository. Such design requires implementing an additional layer of software, and the metadata is stored in separate files. Some studies have suggested encrypting EXIF metadata in image files using symmetric encryption. Wijayanto et al. (2016) proposed a scheme using the eXtended Tiny Encryption Algorithm (XTEA). XTEA cypher is robust and widely used in many cryptographic applications. It can be used in a wide range of hardware due to its low memory requirements and ease of implementation (Andem, 2003). Bhangale (2019) proposed a similar approach to encrypt EXIF metadata with AES encryption. AES is a symmetric encryption algorithm that is well analysed and widely used (Heron, 2009). However, the common weakness of both methods is that symmetric encryption schemes cannot provide authentication and integrity of the data. An attacker can simply copy encrypted EXIF information, modify and even attach it to a different file without the users' knowledge. Also, proposed methods are limited to JPEG files and suffer from the encryption key distribution problem inherent in symmetric encryption schemes. Lepsoy (2015) proposed two (hierarchical and group-based) models for metadata access control. The first model is based on a design where the hierarchical level limits metadata privacy settings. The second model implements the role-based access control where privacy settings of embedded metadata depend on several predefined contextual groups. Both models include the option to remove metadata.
Deleting metadata containing copyright information can lead to copyright infringement and cause significant financial losses for the owner. In 2016, for the practice of eliminating metadata, Facebook was sued by a German photographer. The court agreed to the photographer's claims, and now Facebook will be forced to adapt its system of uploading photos for German users[5]. MIT researchers proposed a metadata-protection scheme called "mix nets" (Kwon, Lu and Devadas, 2020). Mix nets use chains of servers, known as mixes. The limited identity information and shuffling techniques break the link between source and destination, making it difficult for adversaries to obtain information passing over the network. However, this design can expose services, users, and operations in the event of a compromised server. The proposed method encrypts all files, not just metadata, which leads to additional

[4] https://en.wikipedia.org/w/index.php?title=Doxing&oldid=993558752
[5] https://petapixel.com/2016/11/22/german-photographer-sued-facebook-removing-exif-data-won/

server overhead, latency and reduced end-to-end performance. The process provides data in transit security; the data at rest isn't protected. Other studies are focused on user awareness about the information contained in the data. Most of the suggested solutions do not provide sufficient and reliable metadata protection. We conducted our research and concluded that a comprehensive cryptographic approach is needed to achieve security, integrity, and confidentiality of embedded metadata. Further, we will discuss the various cryptography algorithms and the different entities involved in the proposed scheme.

2.2 Study of Encryption Algorithms

Cryptography is the science of methods for ensuring Confidentiality (the impossibility of reading information to outsiders), Integrity (the impossibility of imperceptibly changing information) and Authentication (verifying the authenticity of authorship or other properties of an object). The construction of cryptographic systems is based on the reuse of relatively simple transformations, the so-called cryptographic primitives. Modern cryptography is characterised by the use of open encryption algorithms and computational tools. The encryption algorithms are divided into the following groups:

1) Symmetric encryption is a method that uses the same cryptographic key to encrypt and decrypt information. (Delfs and Knebl, 2007). Depending on the principle of operation, there are two types of symmetric encryption algorithms: block ciphers and stream ciphers. Block algorithms encrypt data in blocks of a fixed length 64, 128 or another number of bits, depending on the algorithm. Modern block ciphers include AES, TEA, XXTEA, Twofish, Blowfish. Stream ciphers involve processing each bit of information by using the corresponding bit of a pseudo-random secret sequence of numbers, formed based on the key and has the same length as the encrypted message. Examples of stream ciphers include RC4, WAKE, HC-256. AES is a symmetric block encryption algorithm, has a block size of 128 bits and a key length of 128/192/256 bits (Heron, 2009). This algorithm is well analysed and widely used. The Tiny Encryption Algorithm (TEA) is a block-type encryption algorithm developed in 1994 at the University of Cambridge by D. Wheeler and R. Needham (Wheeler and Needham, 1995). TEA is a non-proprietary cipher. Due to low memory requirements and ease of implementation, it is widely used in many cryptographic applications and a wide range of hardware. XXTEA (Corrected Block TEA) is an improved version designed to eliminate critical errors and correct weaknesses of the original TEA algorithm. It was presented in a technical report in 1998 (Wheeler J. David and Rojer M. Needham, 1998). XXTEA is executed on simple and fast operations: XOR, substitution, addition (Yarrkov, 2010).

   Symmetric algorithms require fewer resources and demonstrate faster encryption speed than asymmetric algorithms. Most symmetric ciphers are supposedly resistant to attacks by quantum computers, which in theory pose a threat to asymmetric algorithms. The weak point of symmetric encryption is key exchange. The secret key must be transmitted to all parties; however, when transmitted over unsecured channels, it can be intercepted and used by outsiders (Chandra *et al.*, 2014). In our proposed design, we solve this problem by encrypting the key using asymmetric algorithms. Symmetric key algorithms alone cannot provide authentication and integrity.

2) In asymmetric cryptography, keys work in pairs. Each pair consists of public and private keys. If the data is encrypted with a public key, it can only be decrypted with the corresponding private key. Each pair of asymmetric keys is mathematically related. It is impossible to use a public key from one pair and a secret one from another. RSA is a public-key cryptographic system that uses integer factorisation (Rivest, Shamir and Adleman, 1977). It is valid for both encryption and digital signature. The security of RSA is based on the complexity of factoring two large primes. As with any public-key system, each user has two encryption keys: one public and one private. If the sender wants to send a message, he looks for the recipient's public key, encrypts his message with that key, and once the encrypted message reaches the recipient, he takes care of decrypting it with his private key. RSA algorithm is used in a large number of cryptographic applications.

   The ElGamal scheme is a public-key cryptosystem based on the difficulty of computing discrete logarithms on large prime modulo (Menezes, van Oorschot and Vanstone, 1997). The ElGamal cryptosystem includes an encryption algorithm and a digital signature algorithm. Elliptic Curve Cryptography (ECC) is a public key cryptography method based on elliptic curves over finite fields (Hoffstein, Pipher and Silverman, 2008). The significant difference between ECC and RSA or ElGamal algorithms is the key size versus cryptographic strength for comparable security bit levels. ECC provides the exact cryptographic strength of RSA and ElGamal systems but with much smaller keys (Mahto and Yadav, 2017). The Diffie-Hellman protocol allows two or more parties to obtain a shared secret key using matching private and public key pairs (Diffie *et al.*, 1976). The ECDH is an implementation on the Diffie-Hellman algorithm for elliptic curves (Haakegaard and Lang, 2015). Asymmetric key systems are much slower than symmetric key algorithms, requiring longer keys to provide the exact cryptographic strength (Mahajan and Sachdeva, 2013). Public-key cryptographic systems can provide authentication and integrity (Simmons, 1979).

3) A digital signature is a cryptographic scheme designed to identify the originator of the signed message and protect the data from being changed by third parties. Digital signatures are implemented using asymmetric cryptography and cryptographic hash functions. The basis of an electronic digital signature is the mathematical transformation of the data being signed using the signer's private key and the fulfilment of the following conditions:

   • A digital signature can only be created using a private key.

   • Anyone with access to the corresponding public key can check the validity of an electronic digital signature.

   • Any change in the signed data (even a change in just one bit in a large file) invalidates the electronic digital signature.

   Digital signatures are designed to achieve multiple cryptographic goals: data integrity, authentication, and nonrepudiation (Martin, 2012). Digital signature algorithms, in general, consist of three parts:

   - Key Generation Algorithm - generates a random key pair of specific parameters for the user. The two most important aspects are the randomness of the keys, and the key length is consistent with the security level of the algorithm.
   - Digital Signature Algorithm - defines particular ways to apply asymmetric keys to data during the signing process.

- Signature Verification Algorithm - Similarly, signature verification follows a predefined process.

The most common and widely used digital signatures are based on asymmetric encryption algorithms: RSA, DSA, ECDSA, EdDSA and ElGamal Signature Scheme. In asymmetric digital signature schemes, the message is signed using a private key, and verified using a public key. ElGamal signature scheme is rarely used in practice. NSA developed a much more secure algorithm known as Digital Signature Algorithm (DSA), a cryptographic system that uses the private key from a key pair to create an electronic signature (Schneier, 1996). DSA security is based on mathematical concepts of discrete logarithm complexity and modular exponentiation properties. It is not used for encryption like the RSA scheme. RSA is a public-key cryptographic system that uses integer factorisation, and its security is based on the complexity of factoring two large primes. It can be used both for encryption and electronic signatures. ECDSA is a subset of DSA that includes elliptic curve cryptography. It provides a level of security similar to RSA but with much smaller keys (Johnson and Menezes, 1999). Therefore ECDSA is a desirable algorithm for the implementation of digital signatures.

4) A cryptographic hash function is a mathematical function that takes as input a variable-length string and converts it to a fixed, enciphered string, usually called a message digest or a hash value (Halevi and Krawczyk, 2008). Hash functions are mainly used when calculating checksums from data and when creating electronic signatures. The results of hash functions are statistically unique and called hash code, checksum, or message digest. Currently, the most popular cryptographic hash functions are SHA-1, SHA-256 and MD5. MD4 is the fastest hash function optimised for 32-bit machines in the M.D. family, developed by University of Massachusetts professor R. Rivest in 1990 (Rivest, 1990). MD4 contains three loops of 16 steps each. In 1993, the MD4 cracking algorithm was described, so today, this function is not recommended for use with real applications. MD5 is the most common of the M.D. family of functions. It is similar to MD4, but security enhancements make it 33% slower than MD4. MD5 contains four cycles of 16 steps each and provides data integrity control (Ciampa, 2009). Since the first successful attempt to break this hash function dates back to 1993, many researchers showed that the algorithm makes pseudo-collisions possible. MD5 is currently not recommended for use in real-world applications. In 1993, the NSA worked with NIST to develop a hashing algorithm, SHA-1 (published in FIPS PUB 180), for the secure hashing standard (Dang, 2015). It creates a 160-bit value, also called a message digest. This hash function contains four stages. SHA-1 holds more rounds and runs on a bigger buffer than MD5, and runs slower than MD5 on the same hardware (Aggarwal, 2014). SHA-2 is a family of cryptographic hash functions that include the SHA-256 algorithm. As studies have shown, SHA-2 algorithms work 2-3 times slower than MD5 and SHA-1 hash algorithms (Wei Dai, 2009). The SHA-3 hash function (also called Keccak) is a variable bit function developed in 2012 (Swenson, 2012). The SHA-3 function algorithm is built on the principle of a cryptographic sponge construction. In 2015, the function algorithm was approved and published as FIPS 202 (Hernandez, 2015).

3 Research Methodology

In this research paper, we have proposed a combination of symmetric and asymmetric encryption schemes to ensure embedded metadata's confidentiality, integrity, and authentication. To compare and evaluate the effectiveness of the proposed design, we have implemented six various combinations of symmetric and asymmetric algorithms and digital signatures. Further, we will discuss the proposed schemes in pairs. The methods in the pairs are similar in design and differ only in the type of symmetric encryption.

- In methods 1 and 2, we used AES/XXTEA, RSA and RSA digital signature algorithms. For symmetric encryption used AES symmetric block cipher, with a block size of 128 bits and a key length of 128 bits. XXTEA is a block cipher with a 128 bits key length and a block size at least of 64 bits. XXTEA with 128 bits key length and AES with 128 bits key length in CBC mode is used for metadata encryption. RSA is used for symmetric key wrapping, data integrity and user authentication. The security of the RSA algorithm is based on the complexity of factoring two large primes. In the RSA system, each participant has both a public key *pubKey {e,n}* and a private key *privKey {d,n}.* In the proposed scheme, the sender first encrypts the metadata *(M)* with symmetric systems using randomly generated secret key *k:*
  $C = E_k(M)$
  The secret key *k* is then encrypted with an RSA algorithm using the receiver public key:
  $c=R_{pubKey}(k) = m^e \bmod n$
  To decrypt the metadata, receiver first applies RSA using their private key to get the secret key:
  $k = R_{privKey} (c) = c^d \bmod n$
  and then decrypts metadata with a symmetric algorithm using a secret key *k*:
  $M = D_k(C)$
  RSA is also used for digital signatures to ensure data integrity and user authentication. The sender calculates the hash value *h* (using a cryptographic hash function SHA-256) of the file to be signed. The digital signature *g* is created using the sender's private key *s {d,n}*:
  $g = S_s(h) = h^d \bmod n$
  Receiver reads digital signature from the embedded metadata and decrypts it using sender's public key *p {e,n}*:
  $h' = S_p(g) = s^e \bmod n$
  To verify digital signature receiver computes the hash value *h* of the file with ciphertext metadata using SHA-256 and compares it with *h'.*
- In methods 3 and 4, we used AES/XXTEA, ElGamal and DSA digital signature algorithms. ElGamal is a public-key cryptosystem based on the difficulty of computing discrete logarithms on large prime modulo. The ElGamal cryptosystem includes encryption and digital signature algorithms. ElGamal encryption should not be confused with the ElGamal digital signature algorithm, which is rarely used in practice. Therefore in the proposed methods 3 and 4, the ElGamal encryption algorithm is used for symmetric key wrapping, and DSA is used to create digital signatures. In the ElGamal cryptosystem, each party has a public key *{p}* and the private key *{s}* which must be kept secret. For the entire group of subscribers, a large prime number *q* and a number *g* are chosen such that $1 < g$. A pair of private *s* and public keys *p* created as follow:
  $1 <= s <= q-1$

$p=g^s \mod q$
it is assumed that the message is presented as a number $M < q$.
Thus, the plaintext data M is encrypted as follows:
A session key is selected - a random integer k
Knowing receiver’s public key p, the following numbers are calculated:
$a = g^k \mod q$ and $b = p^k M \mod q$
A pair of numbers (a, b) is a ciphertext. Knowing the receiver’s private key $\{s\}$, the original plaintext can be computed by the formula:
$M = b * a^{q-1-s} \mod q$
For data integrity and user authentication, we used DSA public key algorithm. In DSA, the secret key is a number $s \in (0, t)$. The public key is calculated by the formula $d= t^s \mod k$. The public parameters are numbers $\{k, t, g, d\}$ and the secret parameter - the number $\{s\}$.
Message signature is performed according to the following algorithm:
A random number $n \in (0, t)$ is selected, and the following values are calculated
$a = (g^k \mod k) \mod t$
$b = n^{-1}(H(m) + s*a) \mod t$
The DSA signature is a pair (a,b).
Calculations perform signature verification:
$c = b^{-1}$ *mode* t
$e_1 = H(m) * c$ *mode* t
$e_2 = a* c$ *mode* t
$U = (g^{e1} * d^{e2}$ *mode* $k)$ *mode* t
The DSA signature is correct if $U = a$.

- In methods 5 and 6, we used AES/XXTEA, ECDH and ECDSA digital signature algorithms. Instead of generating a random secret password, we implemented an ECDH key exchange protocol to establish a common symmetric key. In ECDH, each party have a key pair consisting of a private key $\{x\}$ and a public key $\{Y\}$. The sender's key pair is $\{x_s, Y_s\}$ and the receiver's key pair is $\{x_r, Y_r\}$. Before executing the protocol, the parties must exchange public keys. Each party calculates common secret key k by multiplying the own private key with the opposite public key:
  $k = x_c \cdot Y_r = x_r \cdot Y_s$
  The sender encrypts the data with symmetric systems using the common secret key k:
  $C=E_k(M)$
  The receiver decrypts data with a symmetric algorithm using a common secret key k:
  $M = D_k(C)$
  We used ECDSA public key algorithm for digital signature verification. ECDSA is similar in structure to DSA but based, in contrast, on the use of elliptic curves cryptography. NIST recommends the use of digital signatures implemented with elliptic curves for asymmetric ECC-based algorithms (Barker and Dang, 2015). The ECDSA signature algorithm takes as input the result of cryptographic hash function H, a private key *privK*. It creates a signature consisting of a pair of integers $\{k, n\}$ as output. The ECDSA signature verification algorithm takes as input a signed message M and the signature $\{k, n\}$ obtained by the signature algorithm using the public key *pubK* corresponding to the signer's private key *privK*.

The proposed design provides hybrid encryption scheme for metadata security by extracting, encrypting and embedding the important tags in the metadata. The symmetric key secrecy is provided by public-key encryption. Only the users with matching key pairs can access encrypted information. Metadata remains embedded in the file and is not stripped from it during storage or transmission. The proposed design ensures confidentiality, integrity, authentication and nonrepudiation of embedded metadata. The above statements are justified by a comparative analysis, calculation and comparison of storage costs and calculation of the proposed methods with modern schemes found in the literature.

4 Design Specification

The encryption algorithms and the corresponding key sizes for proposed methods were chosen in accordance with NIST guidelines and recommendations (Barker and Dang, 2015). Symmetric encryption is implemented using XXTEA and AES ciphers in CBC mode and 128 bits keys. We chose RSA cryptographic algorithm with PKCS padding scheme, ElGamal cryptosystem, and ECDH key exchange protocol for asymmetric encryption. To ensure integrity and authentication, a "Sign-then-Encrypt" scheme was implemented. Initially, we had implemented “Encrypt-then-sign”, but after an additional literature review, we decided to change it as many authors didn’t recommend it. Signing a ciphertext would allow everyone to be able to verify it, not just the receiver. Ciphertext signing in “Encrypt-then-sign” scheme, can also affect nonrepudiation, as the sender may not be aware of the content of the signed ciphertext. Signing a plaintext will allow only the receiver to decrypt and then verify data (Davis, 2001). This design change caused us to redefine all six methods and rewrite the Java code. RSA and DSA algorithms with 2048 bits key and ECDSA with 256 bits key used to generate and verify digital signatures. We chose a 256-bit cryptographic hash function SHA-256 to calculate the hash values. During the encryption/decryption process, embedded metadata is extracted and saved in XMP format. Adobe XMP (Extensible Metadata Platform) is a technology created by Adobe and allows users to add additional information to files and enables the exchange of metadata between different applications[6]. Table 1 below shows combinations for encryption algorithms, corresponding user keys and specifications for the proposed methods.

Table 1. Specifications of the Proposed Methods

Method #	Symmetric Key Algorithm	Symmetric Key Size	Public Key Algorithm	Public Key Type / Size	Digital Signature	Digital Signature Key Type / Size
1	AES	128 bits	RSA	RSA / 2048 bits	RSA	RSA / 2048 bits
2	XXTEA	128 bits	RSA	RSA / 2048 bits	RSA	RSA 2048/ bits

[6] https://www.adobe.com/products/xmp.html

3	AES	128 bits	ElGamal	ElGamal / 2048 bits	DSA	DSA / 2048 bits
4	XXTEA	128 bits	ElGamal	ElGamal / 2048 bits	DSA	DSA / 2048 bits
5	AES	128 bits	ECDH	ECDSA Curve P-256 / 256 bits	ECDSA	ECDSA P-256 / 256 bits
6	XXTEA	128 bits	ECDH	ECDSA Curve P-256 / 256 bits	ECDSA	ECDSA P-256 / 256 bits

Methods 1-4 are similar in design and illustrated in Figures 1 and 2. Methods 5 and 6 are based on Elliptic Curve Cryptography (ECC) and ECDH key exchange protocol and shown in Figures 3 and 4. For the comparable security level ECC requires much shorter keys than algorithms based on modular arithmetic (Maletsky, 2020). The detailed implementation steps included in the proposed methods are as follows:

4.1 Embedded Metadata Encryption Process using AES/XXTEA, RSA/ElGamal and RSA/DSA Digital Signature

1. Input: File with plaintext metadata.
2. Calculate sha-256 hash of the plaintext data
3. Generate a random secret key.
4. Extract metadata and save it to an external XMP file.
5. Encrypt content of XMP file with random password from step 3 using AES or XXTEA algorithms.
6. Remove all metadata from the File.
7. Encrypt secret key with sender's and receivers' public key using RSA or ElGamal algorithms.
8. Embed ciphertext metadata into the File with the following tags: sender's public key, secret key encrypted with the public key of a receiver, secret key encrypted with the public key of a sender, ciphertext metadata.
9. Create and attach RSA or DSA digital signature using sha-256 hash and sender's private key.
10. Output: File with encrypted metadata.

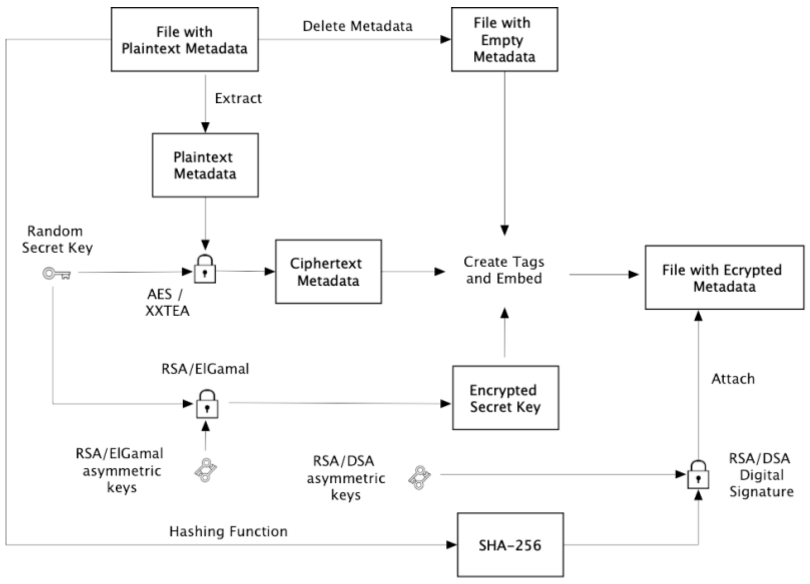


Figure 1: Metadata Encryption Process

4.2 Embedded Metadata Decryption Process Using AES/XXTEA, RSA/ElGamal and RSA/DSA Digital Signature

1 Input: File with encrypted metadata.
2 Read digital signature tag from metadata.
3 Read sender's public key from the metadata.
4 Decrypt secret key with receiver's private key using RSA or Elgamal algorithms.
5 Decrypt ciphertext metadata with a secret key using AES or XTEA algorithms and save it in a separate XML file.
6 Remove metadata from the File.
7 Restore metadata from XML file to the received File (copy all tags from external XML file back to the File).
8 Calculate sha-256 hash of the file
9 Verify the RSA or DSA digital signature using sha-256 hash values and the sender's public key.
10 Output: File with decrypted metadata.

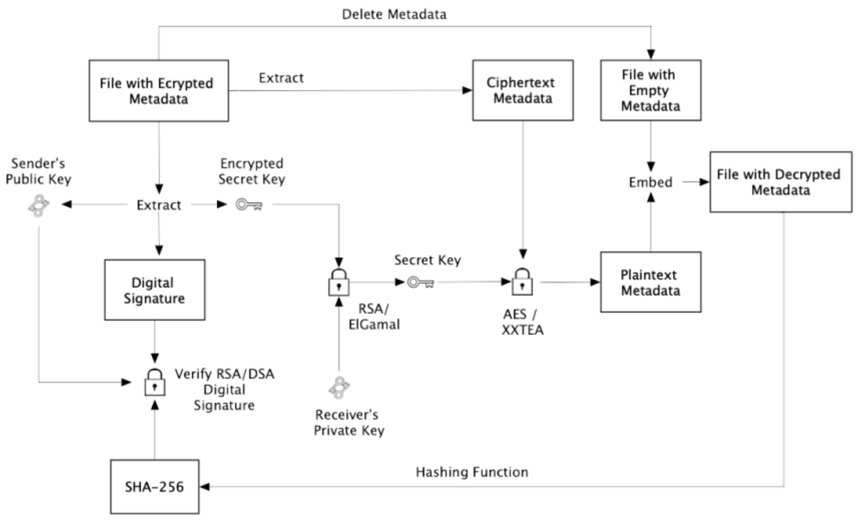


Figure 2: Metadata Decryption Process

4.3 Embedded Metadata Encryption Process using AES/XXTEA, ECDH and ECDSA Digital Signature

1 Input: File with plaintext metadata.
2 Calculate sha-256 hash of the plaintext data
3 Compute common secret key using ECDH key exchange protocol.
4 Extract metadata and save it to an external XMP file.
5 Encrypt content of XMP file with the common secret key from step 3 using AES or XXTEA algorithms.
6 Remove metadata from the File.
7 Create and embed new metadata into the file with the following tags: sender's public key, ciphertext metadata.
8 Create and attach ECDSA digital signature using sha-256 hash and sender's private key.
9 Output: File with encrypted metadata.

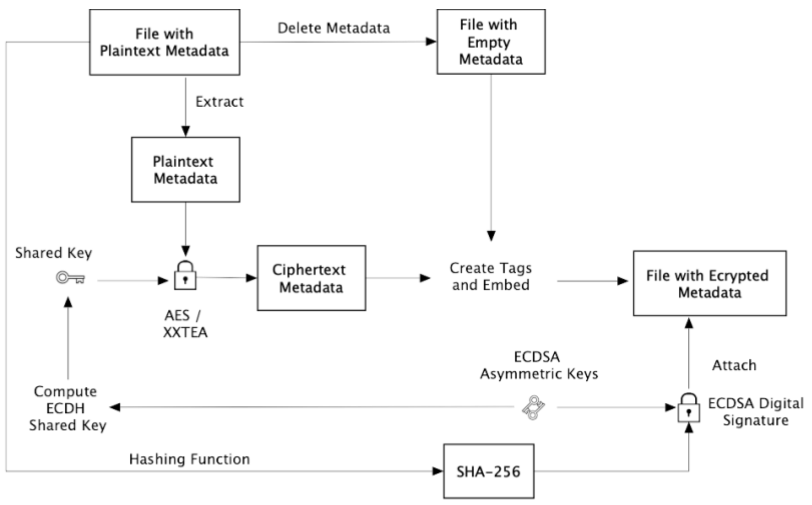


Figure 3: Metadata Encryption Process

4.4 Embedded Metadata Decryption Process using AES/XXTEA, ECDH and ECDSA Digital Signature

1 Input: File with encrypted metadata.
2 Read digital signature tag from metadata.
3 Read sender's public key from the metadata.
4 Compute the common secret key using ECDH key exchange protocol.
5 Decrypt ciphertext metadata with a common secret key using AES or XTEA algorithms and save it in a separate XML file.
6 Remove all metadata from the File.
7 Restore metadata from XML file to the received File (copy all tags from external XML file back to the File).
8 Calculate sha-256 hash of the file

9 Verify ECDSA digital signature using sha-256 hash values and sender's public key.
10 Output: File with decrypted metadata.

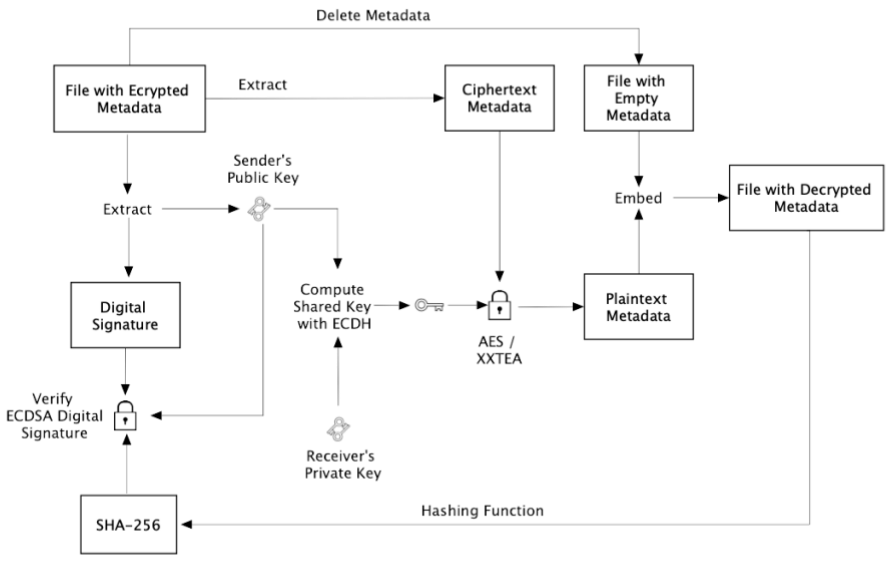


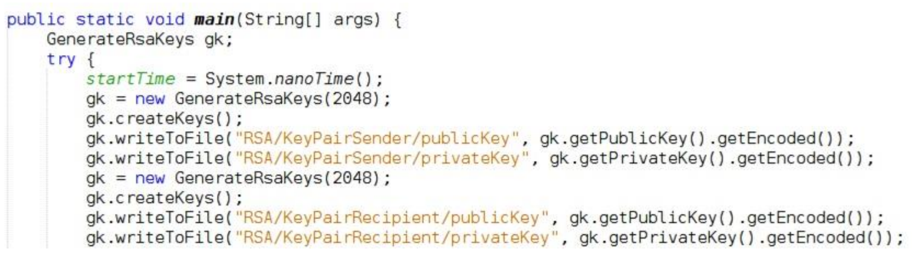
Figure 4: Metadata Decryption Process

5 Implementation

We have implemented the proposed design in Java programming language using the Java security APIs, Bouncy Castle Crypto APIs, Apache common codec. The project code was developed in the NetBeans environment on Linux Virtual Machine. Ubuntu Linux system was installed in VMware Fusion as a guest. Mac OS system was used as a host. A total of six methods were implemented and evaluated. The technical characteristics of the methods are demonstrated in Table 1.

5.1 Keys Generation.

RSA 2048-bits keys are created and placed in specified locations with *GenerateRsaKeys.java* class (Figure 5).



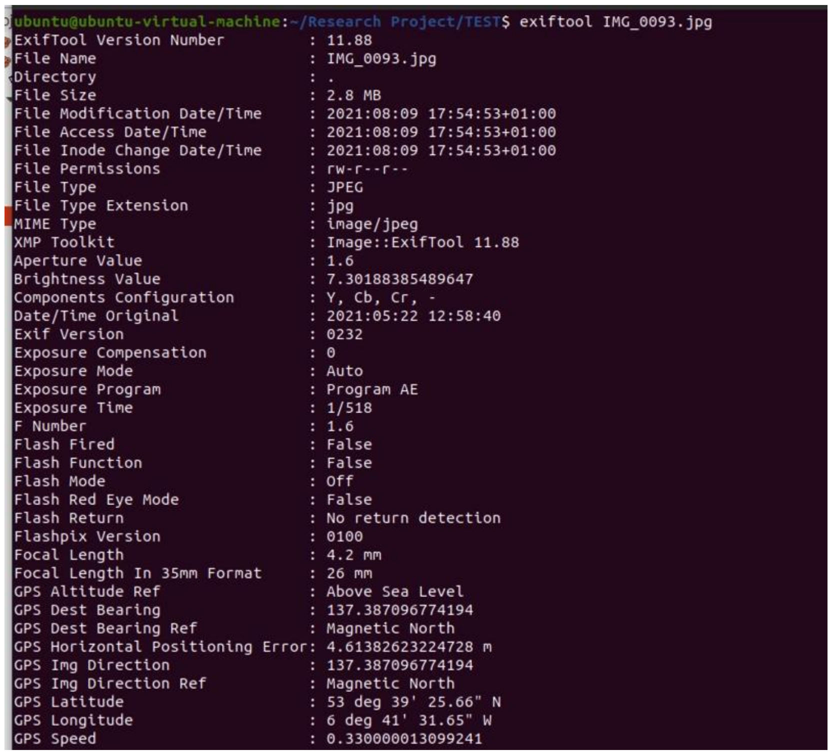
```
public static void main(String[] args) {
    GenerateRsaKeys gk;
    try {
        startTime = System.nanoTime();
        gk = new GenerateRsaKeys(2048);
        gk.createKeys();
        gk.writeToFile("RSA/KeyPairSender/publicKey", gk.getPublicKey().getEncoded());
        gk.writeToFile("RSA/KeyPairSender/privateKey", gk.getPrivateKey().getEncoded());
        gk = new GenerateRsaKeys(2048);
        gk.createKeys();
        gk.writeToFile("RSA/KeyPairRecipient/publicKey", gk.getPublicKey().getEncoded());
        gk.writeToFile("RSA/KeyPairRecipient/privateKey", gk.getPrivateKey().getEncoded());
```

Figure 5: RSA Keys Generation

ElGamal and DSA 2048-bits keys generation is implemented with *GenerateElGamalKeys.java* and *GenerateDsaKeys.java* classes. ECDSA 256-bit keys are created with *GenerateECkeys.java* class.

5.2 Extracting and Removing Embedded Metadata.

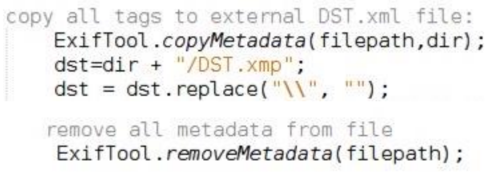
Figure 6 shows the metadata of the Jpeg file.



```
ubuntu@ubuntu-virtual-machine:~/Research Project/TEST$ exiftool IMG_0093.jpg
ExifTool Version Number         : 11.88
File Name                       : IMG_0093.jpg
Directory                       : .
File Size                       : 2.8 MB
File Modification Date/Time     : 2021:08:09 17:54:53+01:00
File Access Date/Time           : 2021:08:09 17:54:53+01:00
File Inode Change Date/Time     : 2021:08:09 17:54:53+01:00
File Permissions                : rw-r--r--
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
XMP Toolkit                     : Image::ExifTool 11.88
Aperture Value                  : 1.6
Brightness Value                : 7.30188385489647
Components Configuration        : Y, Cb, Cr, -
Date/Time Original              : 2021:05:22 12:58:40
Exif Version                    : 0232
Exposure Compensation           : 0
Exposure Mode                   : Auto
Exposure Program                : Program AE
Exposure Time                   : 1/518
F Number                        : 1.6
Flash Fired                     : False
Flash Function                  : False
Flash Mode                      : Off
Flash Red Eye Mode              : False
Flash Return                    : No return detection
Flashpix Version                : 0100
Focal Length                    : 4.2 mm
Focal Length In 35mm Format     : 26 mm
GPS Altitude Ref                : Above Sea Level
GPS Dest Bearing                : 137.387096774194
GPS Dest Bearing Ref            : Magnetic North
GPS Horizontal Positioning Error: 4.61382623224728 m
GPS Img Direction               : 137.387096774194
GPS Img Direction Ref           : Magnetic North
GPS Latitude                    : 53 deg 39' 25.66" N
GPS Longitude                   : 6 deg 41' 31.65" W
GPS Speed                       : 0.330000013099241
```

Figure 6: File with Plaintext Metadata

All operations with embedded metadata are performed using methods of *ExifTool.java* class. Java code of extracting, copying and removing operation is illustrated in Figure 7.

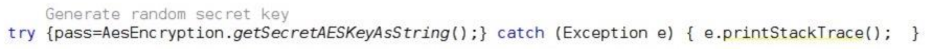


```
copy all tags to external DST.xml file:
    ExifTool.copyMetadata(filepath,dir);
    dst=dir + "/DST.xmp";
    dst = dst.replace("\\", "");

   remove all metadata from file
    ExifTool.removeMetadata(filepath);
```

Figure 7: Metadata Operations

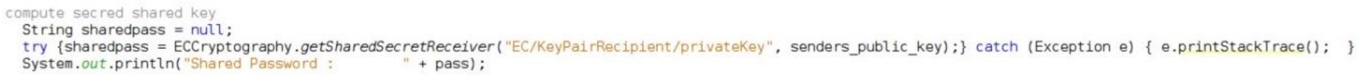
5.3 Metadata Encryption with Symmetric Ciphers.

Before encrypting the metadata, depending on the method, we must generate either a random 128-bit secret key or compute a shared secret key using the ECDH algorithm. For this purpose, we used *AesEncryption.java*, *ECCryptography.java* classes methods (Fig. 8,9).



```
    Generate random secret key
 try {pass=AesEncryption.getSecretAESKeyAsString();} catch (Exception e) { e.printStackTrace();  }
```

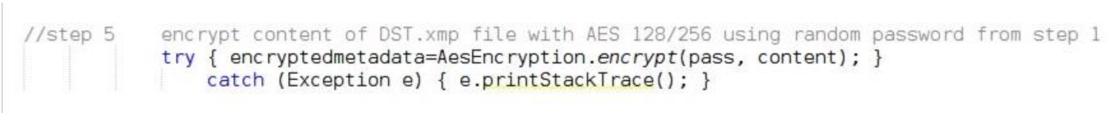
Figure 8: Secret Keys Generation



```
compute secred shared key
  String sharedpass = null;
  try {sharedpass = ECCryptography.getSharedSecretReceiver("EC/KeyPairRecipient/privateKey", senders_public_key);} catch (Exception e) { e.printStackTrace();  }
  System.out.println("Shared Password :        " + pass);
```

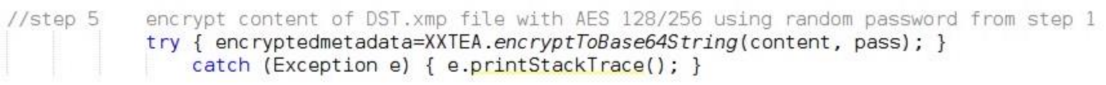
Figure 9: Computing Common Secret Keys using ECDH Protocol

The plaintext is encrypted with AES or XXTEA symmetric algorithms, as shown in Figures 10 and 11.



```
//step 5    encrypt content of DST.xmp file with AES 128/256 using random password from step 1
            try { encryptedmetadata=AesEncryption.encrypt(pass, content); }
                catch (Exception e) { e.printStackTrace(); }
```

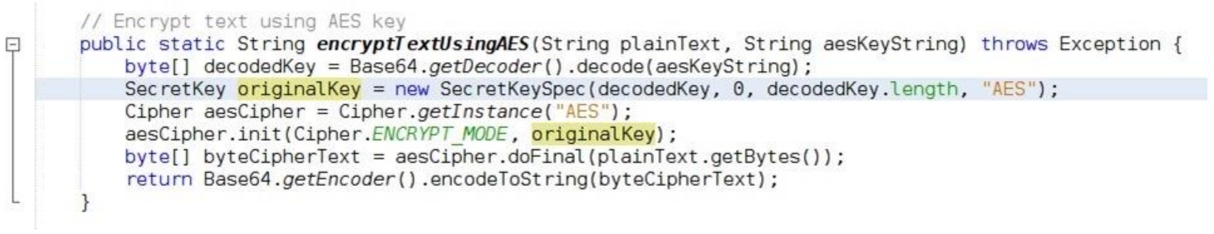
Figure 10: AES Encryption



```java
//step 5    encrypt content of DST.xmp file with AES 128/256 using random password from step 1
            try { encryptedmetadata=XXTEA.encryptToBase64String(content, pass); }
                catch (Exception e) { e.printStackTrace(); }
```

Figure 11: XXTEA Encryption

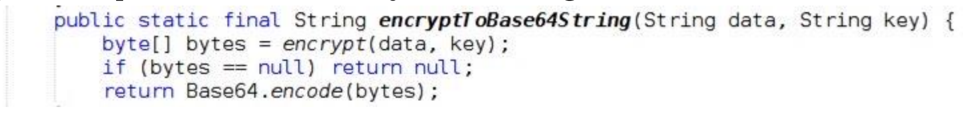
AES encryption is implemented with Java security APIs and is shown in Figure 12.



```java
    // Encrypt text using AES key
    public static String encryptTextUsingAES(String plainText, String aesKeyString) throws Exception {
        byte[] decodedKey = Base64.getDecoder().decode(aesKeyString);
        SecretKey originalKey = new SecretKeySpec(decodedKey, 0, decodedKey.length, "AES");
        Cipher aesCipher = Cipher.getInstance("AES");
        aesCipher.init(Cipher.ENCRYPT_MODE, originalKey);
        byte[] byteCipherText = aesCipher.doFinal(plainText.getBytes());
        return Base64.getEncoder().encodeToString(byteCipherText);
    }
```

Figure 12: AES Encryption Implementation

XXTEA encryption algorithm library for Java is borrowed from GitHub (code author: Ma Bingyao)[7] and implemented in *XXTEA.java* class, Figure 13.

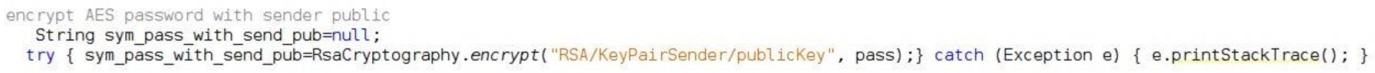


```java
public static final String encryptToBase64String(String data, String key) {
    byte[] bytes = encrypt(data, key);
    if (bytes == null) return null;
    return Base64.encode(bytes);
```

Figure 13: XXTEA Encryption Implementation

5.4 Asymmetric Encryption of Secret Key

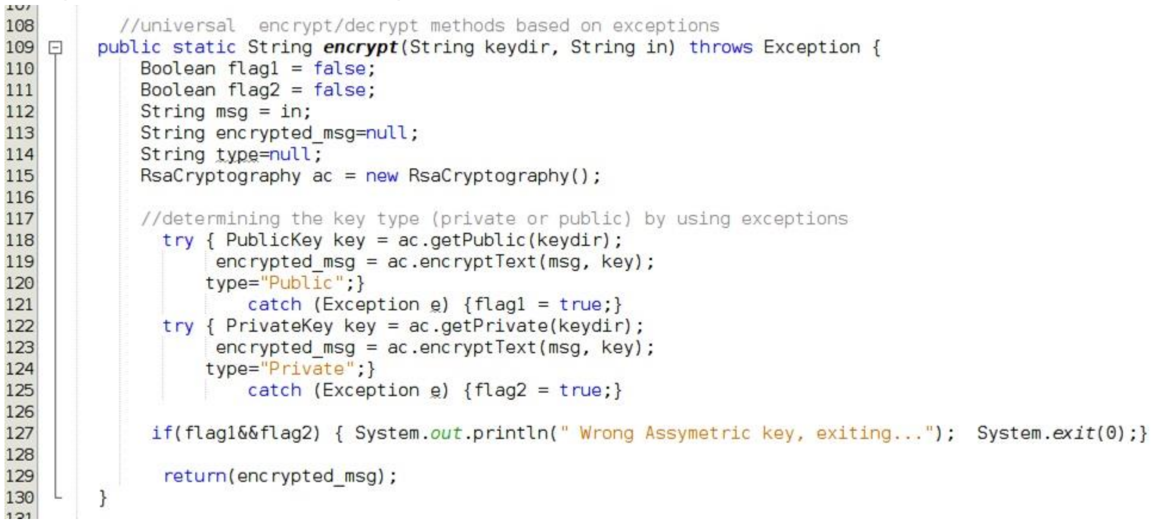
The secret key is then encrypted with a corresponded public key, as shown in Figure 14. The method *encrypts* of *RsaCryptography.java* class takes the path to the public key and plaintext as arguments.



```java
encrypt AES password with sender public
   String sym_pass_with_send_pub=null;
  try { sym_pass_with_send_pub=RsaCryptography.encrypt("RSA/KeyPairSender/publicKey", pass);} catch (Exception e) { e.printStackTrace(); }
```

Figure 14: Secret Key Encryption using RSA

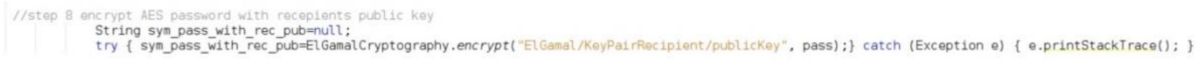
Both *encryption* and *decryption* methods *RsaCryptography.java* class recognise the key type using exceptions, thus reducing the number of methods and code.



```java
      //universal  encrypt/decrypt methods based on exceptions
    public static String encrypt(String keydir, String in) throws Exception {
        Boolean flag1 = false;
        Boolean flag2 = false;
        String msg = in;
        String encrypted_msg=null;
        String type=null;
        RsaCryptography ac = new RsaCryptography();

        //determining the key type (private or public) by using exceptions
          try { PublicKey key = ac.getPublic(keydir);
                encrypted_msg = ac.encryptText(msg, key);
               type="Public";}
                   catch (Exception e) {flag1 = true;}
          try { PrivateKey key = ac.getPrivate(keydir);
                encrypted_msg = ac.encryptText(msg, key);
               type="Private";}
                   catch (Exception e) {flag2 = true;}

         if(flag1&&flag2) { System.out.println(" Wrong Assymetric key, exiting...");  System.exit(0);}

          return(encrypted_msg);
    }
```

Figure 15: Using Exceptions to Determine the Type of the Key

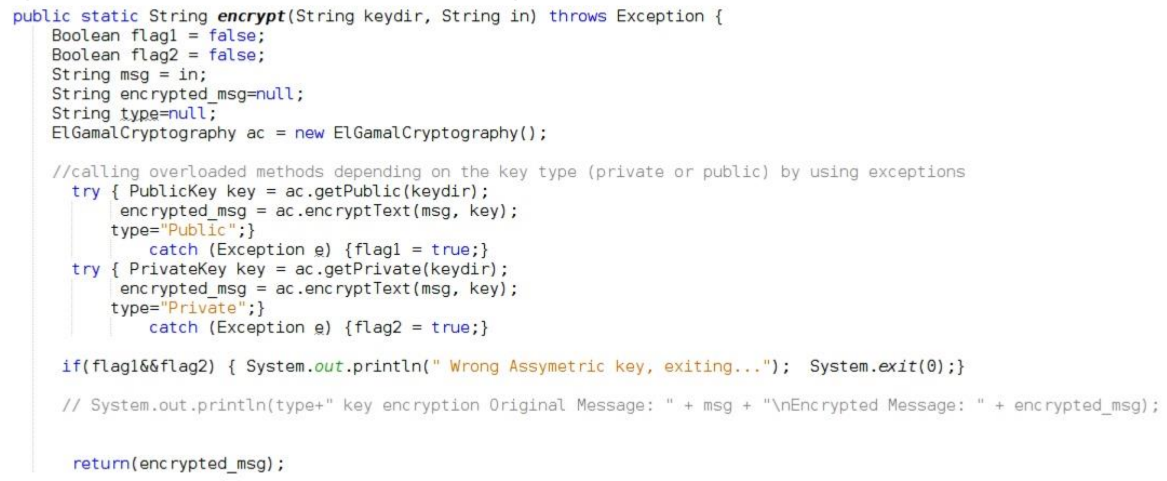
ElGamal encrypation was implemented similarly, as shown in Figures 16 and 17



```java
//step 8 encrypt AES password with recepients public key
        String sym_pass_with_rec_pub=null;
        try { sym_pass_with_rec_pub=ElGamalCryptography.encrypt("ElGamal/KeyPairRecipient/publicKey", pass);} catch (Exception e) { e.printStackTrace(); }
```

[7] https://github.com/xxtea/xxtea-java/blob/master/src/main/java/org/xxtea/XXTEA.java

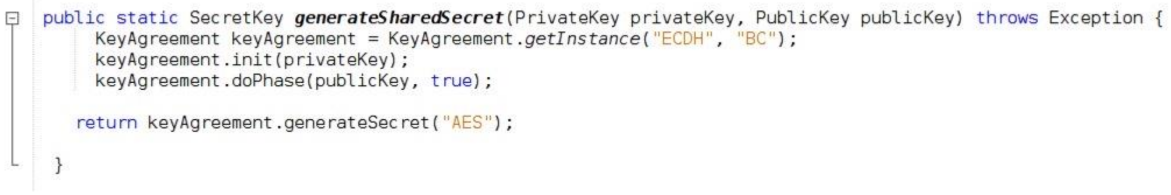
Figure 16: Encryption of Secret Key using ElGamal



```
public static String encrypt(String keydir, String in) throws Exception {
    Boolean flag1 = false;
    Boolean flag2 = false;
    String msg = in;
    String encrypted_msg=null;
    String type=null;
    ElGamalCryptography ac = new ElGamalCryptography();

    //calling overloaded methods depending on the key type (private or public) by using exceptions
      try { PublicKey key = ac.getPublic(keydir);
           encrypted_msg = ac.encryptText(msg, key);
          type="Public";}
              catch (Exception e) {flag1 = true;}
      try { PrivateKey key = ac.getPrivate(keydir);
           encrypted_msg = ac.encryptText(msg, key);
          type="Private";}
              catch (Exception e) {flag2 = true;}

     if(flag1&&flag2) { System.out.println(" Wrong Assymetric key, exiting..."); System.exit(0);}

     // System.out.println(type+" key encryption Original Message: " + msg + "\nEncrypted Message: " + encrypted_msg);


      return(encrypted_msg);
```

Figure 17: ElGamalCryptography.java encrypt Method

There is no need to encrypt a shared key computed with ECDH key exchange protocol in methods 5 and 6, as the key is calculated independently by the receiver. The shared key is computed using the *getSharedSecretSender* method of the *ECCryptography.java* class using the Bouncy Castle Crypto API, Figure 18.

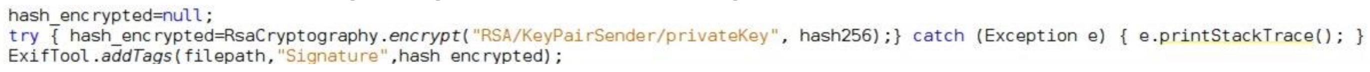


```
public static SecretKey generateSharedSecret(PrivateKey privateKey, PublicKey publicKey) throws Exception {
    KeyAgreement keyAgreement = KeyAgreement.getInstance("ECDH", "BC");
    keyAgreement.init(privateKey);
    keyAgreement.doPhase(publicKey, true);

  return keyAgreement.generateSecret("AES");

}
```

Figure 18: Computing Shared Key Using ECDH Key Exchange Protocol

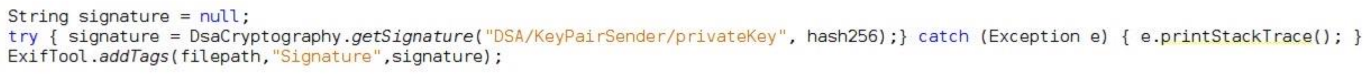
5.5 Digital Signatures Generation

The digital signature creating process depends on the method. The Java code for generating RSA, DSA, and ECDSA digital signatures is shown in Figures 19, 20 and 21.



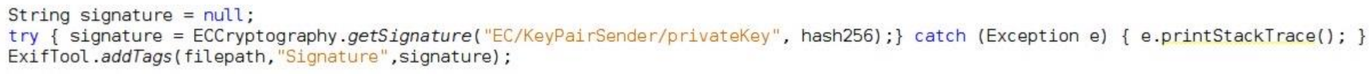
```
hash_encrypted=null;
try { hash_encrypted=RsaCryptography.encrypt("RSA/KeyPairSender/privateKey", hash256);} catch (Exception e) { e.printStackTrace(); }
ExifTool.addTags(filepath,"Signature",hash_encrypted);
```

Figure 19: Generation of RSA Digital Signatures



```
String signature = null;
try { signature = DsaCryptography.getSignature("DSA/KeyPairSender/privateKey", hash256);} catch (Exception e) { e.printStackTrace(); }
ExifTool.addTags(filepath,"Signature",signature);
```

Figure 20: Generation of DSA Digital Signatures

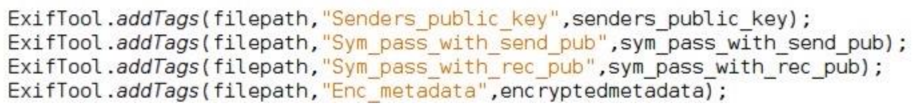


```
String signature = null;
try { signature = ECCryptography.getSignature("EC/KeyPairSender/privateKey", hash256);} catch (Exception e) { e.printStackTrace(); }
ExifTool.addTags(filepath,"Signature",signature);
```

Figure 21: Generation ECDSA Digital Signatures

5.6 Embedding Metadata Tags

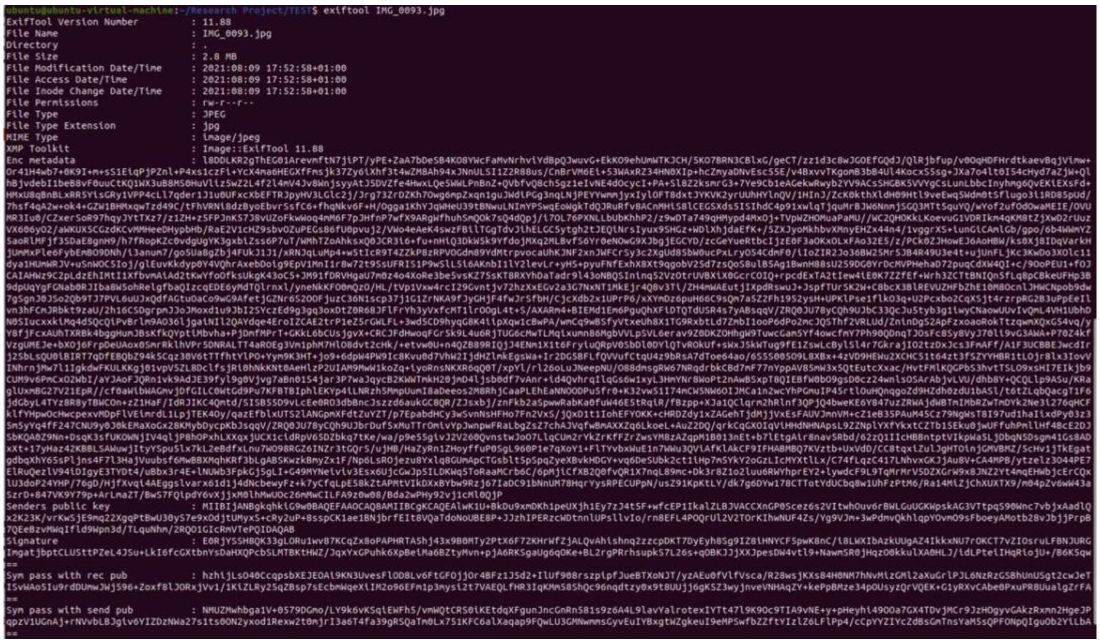
Tags are embedded using *ExifTool.java* class and illustrated in Figure 22.



```
ExifTool.addTags(filepath,"Senders_public_key",senders_public_key);
ExifTool.addTags(filepath,"Sym_pass_with_send_pub",sym_pass_with_send_pub);
ExifTool.addTags(filepath,"Sym_pass_with_rec_pub",sym_pass_with_rec_pub);
ExifTool.addTags(filepath,"Enc_metadata",encryptedmetadata);
```

Figure 22: Embedding Metadata Tags

The file with encrypted metadata and embedded tags is illustrated in Figure 23.

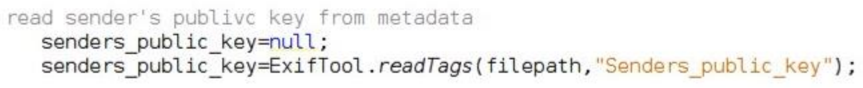


```
ubuntu@ubuntu-virtual-machine:~/Research Project/TEST$ exiftool IMG_0093.jpg
ExifTool Version Number         : 11.88
File Name                       : IMG_0093.jpg
Directory                       : .
File Size                       : 2.8 MB
File Modification Date/Time     : 2021:08:09 17:52:58+01:00
File Access Date/Time           : 2021:08:09 17:52:58+01:00
File Inode Change Date/Time     : 2021:08:09 17:52:58+01:00
File Permissions                : rw-r--r--
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
XMP Toolkit                     : Image::ExifTool 11.88
Enc metadata                    : l8DDLKR2gThEG01ArevmftN7jiPT/yPE+ZaA7bDeSB4KO8YWcFaMvNrhviYdBpQJwuv [illegible]
Or41H4wb7+0K9I+m+sS1EiqPjPZnl+P4xs1czFi+YcX4ma6HEGXfFmsjk37Zy6iXhf3t4wZM8Ah94xJNnULSI1Z2R88us/CnBrVM6 [illegible]
hBjvdebI1beB8vF0uuCtKQ1WX3uB8MS0HuVliz5wZ2L4f2l4mV4Jv8WnjsyyAtJ5DVZfe4HwxLQe5WWLPnBnZ+QVbfvQ8ch5gz1e [illegible]
HMxU8qBnBLxRR5YlsGRy1VPP4cLl7qder1J1u0UFxcXbEFTRJpyHV3LGlc2j/Jrg73ZrDZKh7Owg6mpZxqn1guJWdlPGg3nqLNjP [illegible]
7hsf4qA2w+ok4+GZW1BHMxqwTzd49C/tFhVRNi8dzByoEbvrSsfC6+fhqNkv6F+H/Ogga1KhYJqWHeU39tBNwuLNImYPSwqEoWgkT [illegible]
MR3Iu0/CZxerSoR97hqyJYtTXz7/z1ZH+z5FPJnK57J8vUZoFkwWoq4mM6F7pJHfnP7wfX9ARgWfhuhSmQOk7sQ4dQpj/i7OL76P [illegible]
VX606yO2/aWKUX5CGzdKCvMMHeeDHypbHb/RaE2V1cHZ9sbvOZuPEGs86fU0pvuj2/VWo4eAeK4swzFBilTGgTdvJihELGC5ytgh [illegible]
SaoRlMFjf35DaE8gnH9/h7fRopKZc0vdgUgYK3gxbiZss6P7uT/WMhTZoAhksxQ0JCR3l6+fu+nHlQ3DkW5k9YfdojMXq2MLBvfS [illegible]
jUmMxPle6FybEmBO9DNh/i3anum7/go5Ua8gZbj4FUkJ1J1/xRNJqLuMp4+w5tIcR9T4ZZkP8zRPVOGdm89YdMtrpvocaUhKJNF2 [illegible]
dya1HUmWRJV+uSnWOC5Ioj/glEuvKkdyp0Y4VQhrAxebDoig9EpV1MnI1r8w72t9SsUFRIS1P9wSlL5i6AKnbI1lY2levLr+yHS+ [illegible]
[illegible]
Senders public key              : MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAlwK1U+BkDu9xmDKh1peUXjh1Ey7zJ4t5F+wfcEP1IkalZLBJVACCKnGP0Scez6s2VItwhOuv6rBWLGuUGKWpskAG3VTtpqS90Wnc7vbjxAadlQ
x2K23K/vrKwSjE9mq22XgqPtBwU30yS7e9xOdjtUMyxS+cRy2uP+8sspCK1ae1BNjbrfEIt0YQaTdoNoUBE8P+JJzhIPERzcWDtnnlUPsllvIo/rn8EFL4POQrUl2V2TOrKIhwNUF4Zs/Yg9VJm+3wPdmvQkhlqpYOvmO9sFboeyAMotb28vJbjjPrpB
7QEeBzvMWqIfld9Wpn3d/TLquNhn/2RQO1GIcRmVTePQIDAQAB
Signature                       : E0RjY5SHBQK33gLORu1wv87KCqZx8oPAPHRTA5hj43x9B0MTy2PtX6F72KHrWfZjALQvAhlshnq2zzcpDKT7DyEyh8Sg9IZ8LHNYCF5pwK8nC/L8LWXIbAzkUUgAZ4IkkxNU7rOKCT7vZIOsruLFBNJURG
Imgatjbptc LUSttPZeL4JSu+LkI6fcGXtbnYsDaHXQPcbSLMT8KtHWZ/JqxYxGPuhk6Xp8eLMa6BZtyMvn+pJA6RKSgaUg6qOKe+BL2rgPRrhsupk57L265+qOBKJJ3XXJpesDW4vtl9+NawmSR0jHqzO0kkulXA0HLJ/idLPteiIHqRlojU+/B6KSqw
==
Sym pass with rec pub           : hzhijLsO40CcqpsbXEJEOAl9KN3UvesFlOD8Lv6FtGFOjjOr4BFz1J5d2+Ilufl90rszplpfJueBTXoNJT/yzAEu0fVlfVsca/R28wsjKXs84H0NM7hNvMlzGMl2aXuGrlPJL6NzRzGS8hUnUSgt2cwJeT
ISWWAoSIu9rdDUmwJWjS96+ZoxfBlJ0RxjVv1/1KiZLRy25qZBsp7sEcbmWqeXlIM2o96EFmIp3mysi2t7VAEQLfHR3IqKMm58hQc96nqdtzy0x9t8UUJj6gKSZ3wyjnveVNHAqZY+kePpBMze34pOUsyzQrVQEK+G1yRXvCAbe0PxuPR8UualgZrFA
==
Sym pass with send pub          : NMUZMwhbga1V+0579DGmo/LY9k6vKSqlEWFh5/vmWQtCRS0iKEtdqXFgunJncGnRn581s9z6A4L9lavYalrotexIYTt47l9K9Oc9TIA9vNE+y+pHeyhl490Oa7GX4TDvjMCr9JzHOgyvGAkzRxmn2HgeJP
qpzV1UGnAj+rNVvbLBJglv6YIZDzNWa27s1ts0ON2yxod1Rexw2t0njrI3a6T4fa39gRSQaTm0Lx751KFC6alXaqap9FQwU1UGMNwmnsGyvEuIYBxgtWZgkeuI9eMPSwfbZZftYIzlZ6LFlPp4/cCpYYZIYcZdBsGnTnsYaMSsQPFONpQIguOb2YiLbA
==
```

Figure 23: The File with Encrypted Metadata

5.7 Extracting Metadata Tags

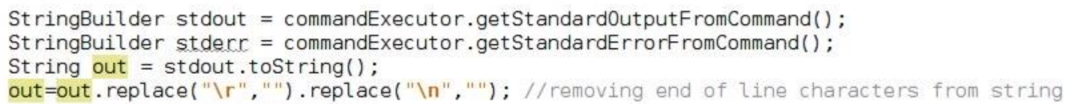
The receiver obtains metadata tags using the *readTags* method of *ExifTool.java* class



```
read sender's publivc key from metadata
   senders_public_key=null;
   senders_public_key=ExifTool.readTags(filepath,"Senders_public_key");
```

Figure 24: Extracting Metadata Tags

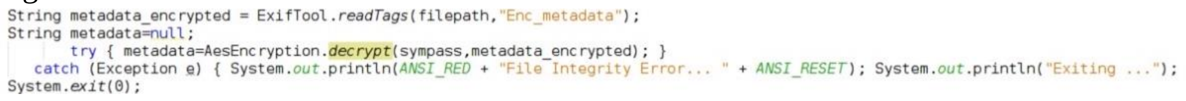
When implementing the *readTags* method, we encountered compiling errors in *AesEncryption.decrypt* method. We fixed this problem by removing the end of line characters from the return string, added by the *SystemCommandExecutor* class methods:



```
StringBuilder stdout = commandExecutor.getStandardOutputFromCommand();
StringBuilder stderr = commandExecutor.getStandardErrorFromCommand();
String out = stdout.toString();
out=out.replace("\r","").replace("\n",""); //removing end of line characters from string
```

Figure 25: Compiler Error Fix

We've also added an exception to the core methods to catch this error to prevent MiTM data integrity attacks if they are deliberately introduced by an attacker by modifying metadata tags:

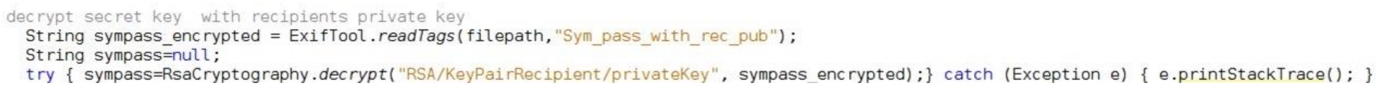


```
String metadata_encrypted = ExifTool.readTags(filepath,"Enc_metadata");
String metadata=null;
        try { metadata=AesEncryption.decrypt(sympass,metadata_encrypted); }
    catch (Exception e) { System.out.println(ANSI_RED + "File Integrity Error... " + ANSI_RESET); System.out.println("Exiting ...");
System.exit(0);
```

Figure 26: Catching Exception Event

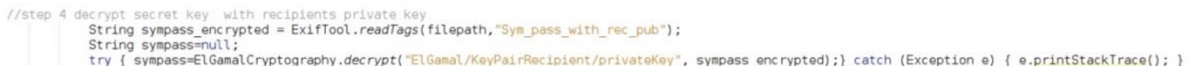
5.8 Decrypting of the Secret Key With Asymmetric Ciphers

Secret key is decrypted with the corresponded private key depending on the method or computed using the ECDH algorithm.



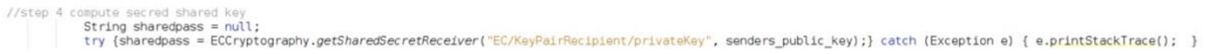
```
decrypt secret key  with recipients private key
  String sympass_encrypted = ExifTool.readTags(filepath,"Sym_pass_with_rec_pub");
  String sympass=null;
  try { sympass=RsaCryptography.decrypt("RSA/KeyPairRecipient/privateKey", sympass_encrypted);} catch (Exception e) { e.printStackTrace(); }
```

Figure 27: Decrypting of Secret Key using RSA



```
//step 4 decrypt secret key  with recipients private key
        String sympass_encrypted = ExifTool.readTags(filepath,"Sym_pass_with_rec_pub");
        String sympass=null;
        try { sympass=ElGamalCryptography.decrypt("ElGamal/KeyPairRecipient/privateKey", sympass encrypted);} catch (Exception e) { e.printStackTrace(); }
```

Figure 28: Decrypting of Secret Key using ElGamal

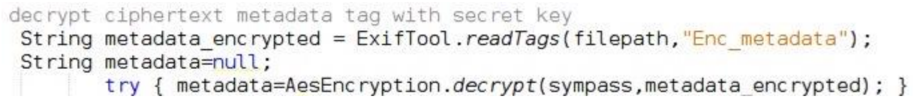


```
//step 4 compute secred shared key
        String sharedpass = null;
        try {sharedpass = ECCryptography.getSharedSecretReceiver("EC/KeyPairRecipient/privateKey", senders_public_key);} catch (Exception e) { e.printStackTrace();  }
```

Figure 29: Computing Common Secret Key using ECDH

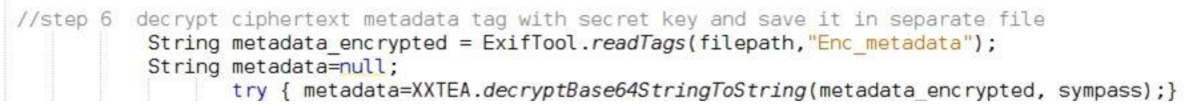
5.9 Decrypting of Ciphertext Metadata with Symmetric Ciphers

Cyphertext is decrypted with AES or XXTEA ciphers.



```
decrypt ciphertext metadata tag with secret key
 String metadata_encrypted = ExifTool.readTags(filepath,"Enc_metadata");
 String metadata=null;
        try { metadata=AesEncryption.decrypt(sympass,metadata_encrypted); }
```

Figure 30: Decrypting Ciphertext Metadata using AES

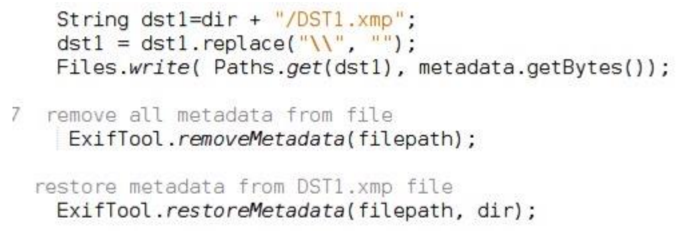


```
//step 6  decrypt ciphertext metadata tag with secret key and save it in separate file
          String metadata_encrypted = ExifTool.readTags(filepath,"Enc_metadata");
          String metadata=null;
                try { metadata=XXTEA.decryptBase64StringToString(metadata_encrypted, sympass);}
```

Figure 31: Decrypting Ciphertext Metadata using XXTEA

5.10 Embedding Decrypted Metadata

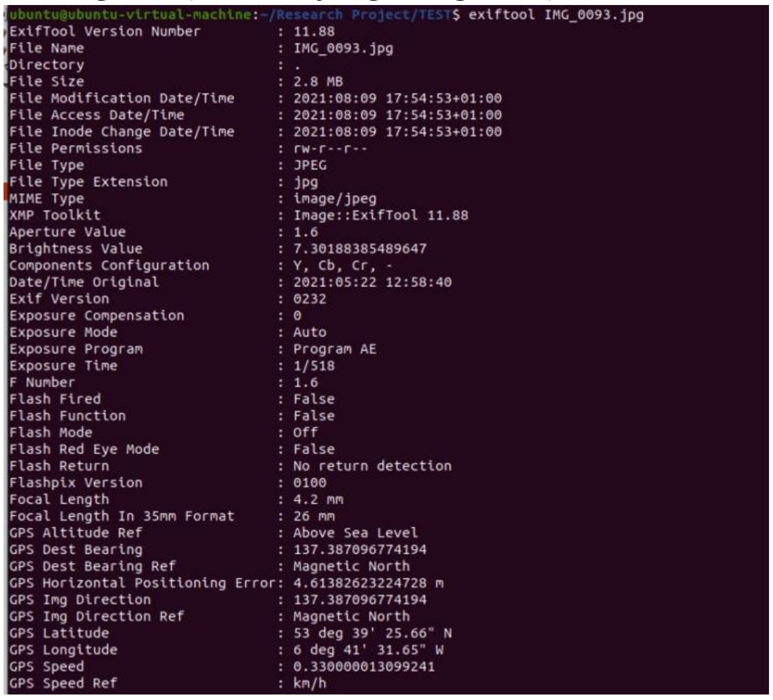
To embed decrypted metadata, we first save it in a separate XMP file, remove it from the original file and then restore it from the XMP file using *ExifTool.java* class methods, as illustrated in Figure 32



```
String dst1=dir + "/DST1.xmp";
dst1 = dst1.replace("\\", "");
Files.write( Paths.get(dst1), metadata.getBytes());

7  remove all metadata from file
    ExifTool.removeMetadata(filepath);

   restore metadata from DST1.xmp file
    ExifTool.restoreMetadata(filepath, dir);
```

Figure 32: Embedding Decrypted Metadata

The output file with decrypted metadata is illustrated in Figure 33, which is the same as the original file shown in Figure 6 (verified by digital signature).

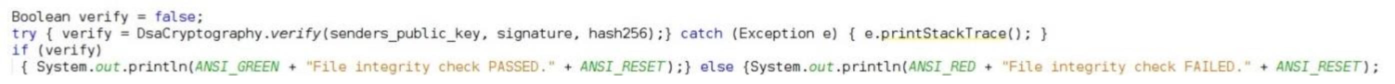


```
ubuntu@ubuntu-virtual-machine:~/Research Project/TEST$ exiftool IMG_0093.jpg
ExifTool Version Number         : 11.88
File Name                       : IMG_0093.jpg
Directory                       : .
File Size                       : 2.8 MB
File Modification Date/Time     : 2021:08:09 17:54:53+01:00
File Access Date/Time           : 2021:08:09 17:54:53+01:00
File Inode Change Date/Time     : 2021:08:09 17:54:53+01:00
File Permissions                : rw-r--r--
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
XMP Toolkit                     : Image::ExifTool 11.88
Aperture Value                  : 1.6
Brightness Value                : 7.30188385489647
Components Configuration        : Y, Cb, Cr, -
Date/Time Original              : 2021:05:22 12:58:40
Exif Version                    : 0232
Exposure Compensation           : 0
Exposure Mode                   : Auto
Exposure Program                : Program AE
Exposure Time                   : 1/518
F Number                        : 1.6
Flash Fired                     : False
Flash Function                  : False
Flash Mode                      : Off
Flash Red Eye Mode              : False
Flash Return                    : No return detection
Flashpix Version                : 0100
Focal Length                    : 4.2 mm
Focal Length In 35mm Format     : 26 mm
GPS Altitude Ref                : Above Sea Level
GPS Dest Bearing                : 137.387096774194
GPS Dest Bearing Ref            : Magnetic North
GPS Horizontal Positioning Error: 4.61382623224728 m
GPS Img Direction               : 137.387096774194
GPS Img Direction Ref           : Magnetic North
GPS Latitude                    : 53 deg 39' 25.66" N
GPS Longitude                   : 6 deg 41' 31.65" W
GPS Speed                       : 0.330000013099241
GPS Speed Ref                   : km/h
```

Figure 33: File with Decrypted Metadata

5.11 Digital Signature Verification

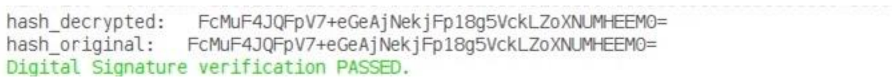
Digital signature verification depends on the method; an example of Java code used for DSA signature verification is demonstrated in Figure 34.



```
Boolean verify = false;
try { verify = DsaCryptography.verify(senders_public_key, signature, hash256);} catch (Exception e) { e.printStackTrace(); }
if (verify)
 { System.out.println(ANSI_GREEN + "File integrity check PASSED." + ANSI_RESET);} else {System.out.println(ANSI_RED + "File integrity check FAILED." + ANSI_RESET);
```

Figure 34: DSA Digital Signature Verification

The software output is illustrated in Figure 35, which shows verification status (passed or failed).



```
hash_decrypted:   FcMuF4JQFpV7+eGeAjNekjFp18g5VckLZoXNUMHEEM0=
hash_original:   FcMuF4JQFpV7+eGeAjNekjFp18g5VckLZoXNUMHEEM0=
Digital Signature verification PASSED.
```

Figure 35: Digital Signature Verification Output

6 Evaluation

In the following sections, we evaluate and compare the results of the proposed methods. The assessment focuses on analysing the performance of the implemented methods, including encryption key generation time, memory consumption, encryption/decryption time, and data capacity analysis. The focus on these results is essential as execution time and memory consumption correspond to the computing device's hardware requirements and power consumption.

6.1 Generation of Asymmetric Key Pairs

To evaluate average keys generation times, we have created Java outer classes to iterate through the main methods specified number of times.

The results are demonstrated in Table 2 and Figure 36.

Table 2. Key Pair Generation Times

Key Pair Type/Bits	Average Key Generation Time, Milliseconds
RSA 2048	542
ElGamal 2048	181
ECDSA 256	36
DSA 2048	26

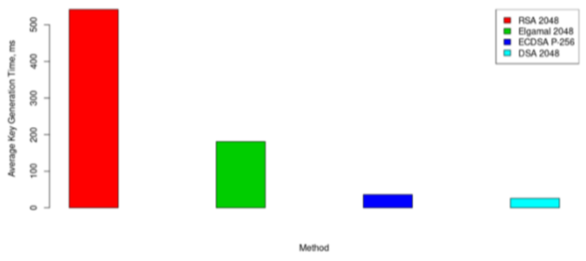


Figure 36: Comparative Status of Key Pairs Generation Times

RSA keys pair generation is the slowest, and DSA is the fastest.

6.2 Average Encryption Time Analysis

We created outer Java classes to iterate through the main classes to evaluate the time required to encrypt/decrypt metadata for each method. Results are demonstrated in Tables 3 and 4, Figures 37 and 38. Detailed descriptions and specifications for each technique are summarised in Table 1.

Table 3. Metadata Encryption Times

Method #	File Type	Average Encryption Time, Milliseconds
1	Jpeg	2258
2	Jpeg	2260
3	Jpeg	2396
4	Jpeg	2397
5	Jpeg	2114
6	Jpeg	2132

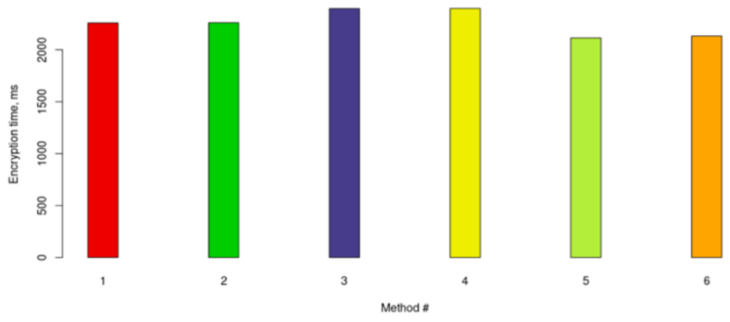


Figure 37: Comparative Status of Encryption Times

6.3 Average Decryption Time Analysis

Table 4. Metadata Decryption Times analysis

Method #	File Type	Average Decryption Time, Milliseconds
1	Jpeg	823
2	Jpeg	826
3	Jpeg	831
4	Jpeg	838
5	Jpeg	738
6	Jpeg	745

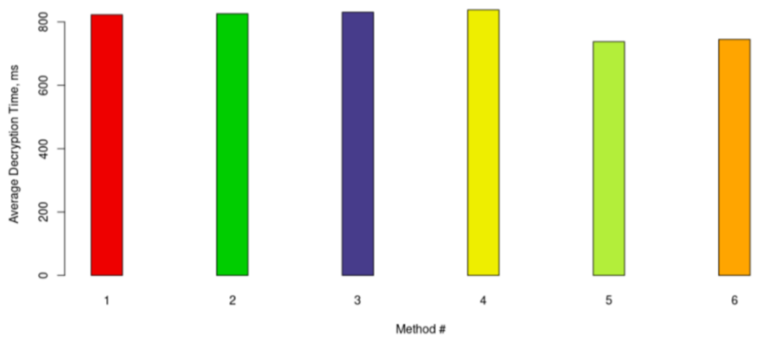


Figure 38: Comparative Status of Decryption Times

6.4 Memory Utilisation Analysis

The comparative status of memory utilisation for each method is demonstrated in Table 5 and Figure 39.

Table 5. Memory utilisation

Method #	File Type	Memory Utilised by the Program, MB
1	Jpeg	15.24
2	Jpeg	15.07
3	Jpeg	25.48
4	Jpeg	25.29
5	Jpeg	47.60
6	Jpeg	42.78

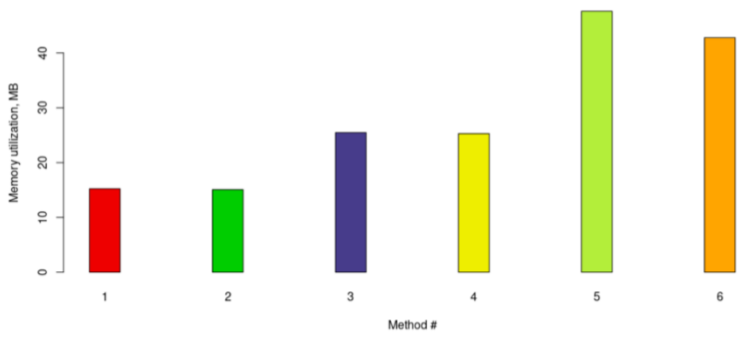


Figure 39: Comparative Status of Memory Utilization

6.5 Data Capacity Analysis

The file with encrypted metadata increases in size by including extra tags, as demonstrated in Table 6 and Figure 40.

Table 6. File Size Increase Results

Method #	File Type	Original File Size, MB	Increase in File Size, Bytes / %
1	Jpeg	7.1	50595.0 / 0.045
2	Jpeg	7.1	50715.0 / 0.046
3	Jpeg	7.1	73155.0 / 0.065
4	Jpeg	7.1	73275.0 / 0.066
5	Jpeg	7.1	30765.0 / 0.028
6	Jpeg	7.1	30885.0 / 0.028

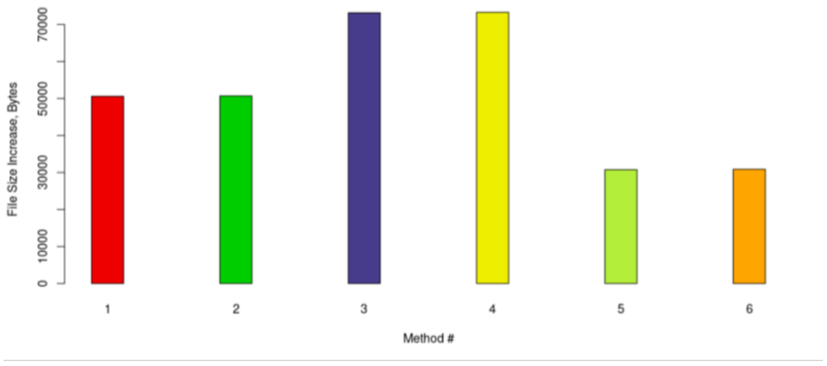


Figure 40: File size increase comparison

6.6 Data Integrity (Digital Signature Verification) Analysis

Previously proposed metadata protection methods, based on symmetric encryption, are generally insecure. An adversary can alter the contents of a file or perform a ciphertext attack without the user's knowledge. Methods proposed in this paper include digital signature verification to prevent such attacks. Any manipulation of the file's content or ciphertext attack will be detected and reported to the user. We tested our methods against data integrity attacks, and all attempts to alter the data were successfully detected. For example, if we crop the Jpeg file (Figure 41), the software will detect such an attack by verifying the digital signature, inform the user and exit (Figure 42).

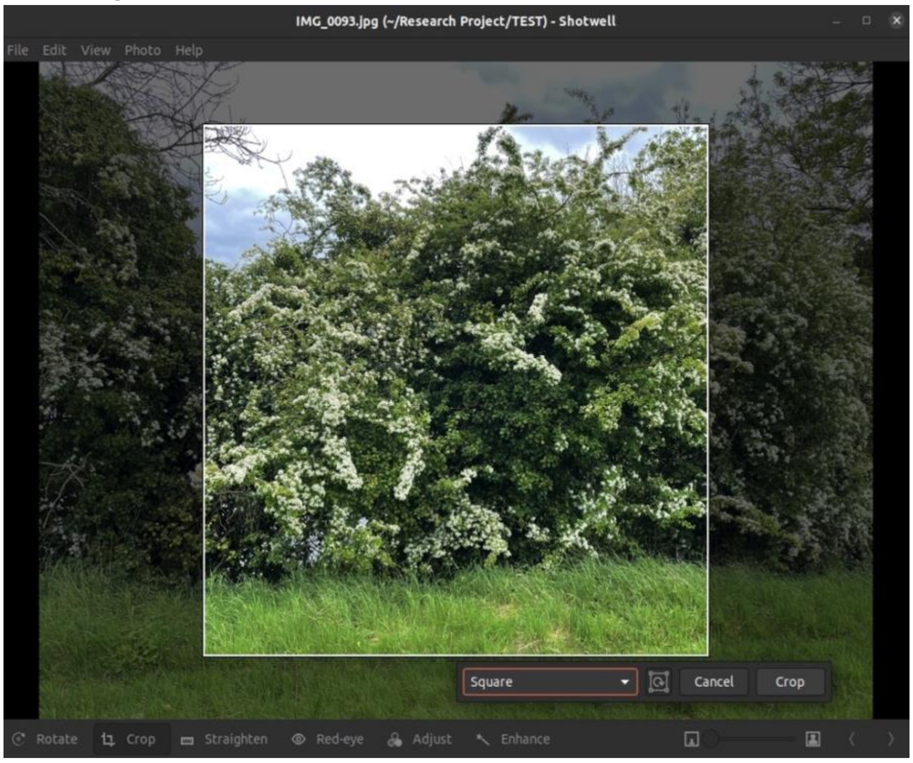
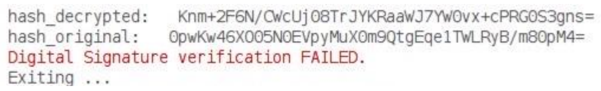

IMG_0093.jpg (~/Research Project/TEST) - Shotwell
File Edit View Photo Help
Square
Cancel
Crop
Rotate
Crop
Straighten
Red-eye
Adjust
Enhance

Figure 41: Modifying an Image File by Cropping



```
hash_decrypted:   Knm+2F6N/CWcUj08TrJYKRaaWJ7YW0vx+cPRG0S3gns=
hash_original:   0pwKw46XO05N0EVpyMuX0m9QtgEqe1TWLRyB/m80pM4=
Digital Signature verification FAILED.
Exiting ...
```

Figure 42: Digital Signature Verification

6.7 Comparative Analysis of Different Approaches to Metadata Protection

Previous approaches for securing metadata are compared and summarized in Table 7. Unlike other methods, only the model proposed in this research paper provides for the confidentiality, integrity, and authentication of embedded metadata.

Table 7. Approach Evaluation

Paper	Approach	Pros	Cons	Confidentiality	Integrity	Authentication	Nonrepudiation
Encryption EXIF Metadata for Protection Photographic Image of Copyright Piracy	Encrypting EXIF metadata with XTEA symmetric cipher	Provides confidentiality	Only Jpeg files are supported, vulnerable to MiTM and data integrity attacks. Key management issues	**Yes**	**No**	**No**	**No**
Securing Image Metadata using Advanced Encryption Standard	Encrypting EXIF metadata with AES symmetric cipher	Easy to implement, provides confidentiality	Only Jpeg files are supported, vulnerable to MiTM and data integrity attacks. Key management issues	**Yes**	**No**	**No**	**No**
Metadata protection scheme for JPEG privacy security using hierarchical and group-based models	Hierarchical and group-based models for metadata access control	Has metadata access control based on privacy policies	This concept requires a third party to implement, doesn't provide confidentiality	**Yes**	**No**	**No**	**No**
A study on image security in social media using digital watermarking with metadata	Embedding EXIF information into a particular image through the digital watermarking algorithms	Protects the metadata from removal by users or social media	Only works with image files, vulnerable to data integrity attacks, image compression algorithms can still damage a watermark	**No**	**No**	**No**	**No**
Proposed Paper	Secure embedded metadata with symmetric and asymmetric encryption	Provides Confidentiality, Data Integrity and User Authentication. Supports many file types	Require users to set up public/private keys	**Yes**	**Yes**	**Yes**	**Yes**

2.1 Discussion

We have implemented, tested and evaluated the performance of the proposed design using different combinations of symmetric/asymmetric cryptography schemes and digital signatures. Conducted tests showed that the RSA key generation time was the slowest, followed by ElGamal and ECDSA, and the DSA key generation time was the fastest. RSA and ElGamal take longer to generate 2048 bit keys because the calculation must include a modular expression. The encryption and decryption times of RSA-based methods 1 and 2 are better than ElGamal-based methods 3 and 4 (Siahaan and Oktaviana, 2018), while methods 5,6 based on elliptic curve key-exchange algorithms were the fastest. AES based methods 1,3,5 were slightly quicker than XXTEA methods 2,4,6 but required more memory to execute. Methods 5 and 6 based on elliptic curve algorithms outperformed all other methods in encryption and decryption speed (Mahto and Yadav, 2017), but they utilise more memory than other methods. Encrypted file increases in size because of extra tags added. Methods 3 and 4 based on the ElGamal algorithm produced the largest files, followed by RSA and ECDH based methods.

Methods 5 and 6 had the smallest files. Drawing from the results above, we can conclude that methods 5 and 6, based on elliptic curve cryptography, are the fastest performing algorithms across all metrics except memory utilisation. Choosing the correct method in the proposed design is a trade-off between execution time and memory consumption.

7 Conclusion and Future Work

In many cases, embedded metadata contains sensitive and essential information about the original data. As recent years have shown, the use of insecure metadata has led to severe hacking incidents and many security breaches, so the metadata information must be protected. In this report, we proposed six metadata protection methods and implemented them in the Java programming language. The proposed design ensures confidentiality, integrity, authentication and nonrepudiation of embedded metadata. None of the approaches described in previous studies provides metadata integrity, authentication, and nonrepudiation. The proposed design can be used in sensitive or confidential environments such as military, pharmaceutical, medical and legal applications and to comply with GDPR regulations. However, encryption makes the metadata information unavailable for processing. A possible solution is to use attribute-based searchable encryption (ABSE) schemes. ABSE technique enables detailed search and retrieval of data files using the encrypted metadata without disclosing any information in plaintext (Chaudhari and Das, 2021). Future work will be to look at ways of combining the proposed design with the core ABSE techniques, as this could enhance the usability and interoperability of the proposed methods.

References

Aggarwal, S. (2014) ‘A review of Comparative Study of MD5 and SHA Security Algorithm’, *International Journal of Computer Applications*, 104, p. 4.

Al-Marashi, I. (2006) ‘The “Dodgy Dossier:” The Academic Implications of the British Government’s Plagiarism Incident’, *Middle East Studies Association Bulletin*, 40(1), pp. 33–43.

Andem, V. R. (2003) ‘A Cryptoanalysis of the Tiny Encryption Algorithm’, p. 68.

Bane, A. and Minnear, R. (2017) ‘Method and system for the normalisation, filtering and securing of associated metadata information on file objects deposited into an object store’. Available at: https://patents.google.com/patent/US9619487B2/en (Accessed: 25 October 2020).

Barker, E. B. and Dang, Q. H. (2015) *Recommendation for Key Management Part 3: Application-Specific Key Management Guidance*. NIST SP 800-57Pt3r1. National Institute of Standards and Technology, p. NIST SP 800-57Pt3r1. doi: 10.6028/NIST.SP.800-57Pt3r1.

Bhangale, R. (2019) ‘Securing Image Metadata using Advanced Encryption Standard’

Chandra, S. *et al.* (2014) ‘A comparative survey of Symmetric and Asymmetric Key Cryptography’, in *2014 International Conference on Electronics, Communication and Computational Engineering (ICECCE). 2014 International Conference on Electronics,* *Communication and Computational Engineering (ICECCE)*, pp. 83–93. doi: 10.1109/ICECCE.2014.7086640.

Chaudhari, P. and Das, M. L. (2021) ‘Privacy Preserving Searchable Encryption with Fine-Grained Access Control’, *IEEE Transactions on Cloud Computing*, 9(2), pp. 753–762. doi: 10.1109/TCC.2019.2892116.

Ciampa, M. D. (2009) *CompTIA Security+ 2008 in depth*. Australia ; United States : Course Technology/Cengage Learning. Available at: http://archive.org/details/comptiasecurity20000ciam (Accessed: 15 June 2021).

Dang, Q. H. (2015) *Secure Hash Standard*. NIST FIPS 180-4. National Institute of Standards and Technology, p. NIST FIPS 180-4. doi: 10.6028/NIST.FIPS.180-4.

Davis, D. (2001) *Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML*. Available at: https://theworld.com/~dtd/sign_encrypt/sign_encrypt7.html (Accessed: 10 July 2021).

Delfs, H. and Knebl, H. (2007) *Introduction to Cryptography: Principles and Applications*. Springer Science & Business Media.

Diffie *et al.* (1976) ‘New Directions in Cryptography’, pp. 644–654.

Faiz bin Jeffry, M. A. and Mammi, H. K. (2017) ‘A study on image security in social media using digital watermarking with metadata’, in *2017 IEEE Conference on Application, Information and Network Security (AINS)*, pp. 118–123. doi: 10.1109/AINS.2017.8270435.

Halevi, S. and Krawczyk, H. (2008) *Randomized Hashing and Digital Signatures*. Available at: https://www.ee.technion.ac.il/~hugo/rhash/ (Accessed: 15 June 2021).

Hernandez, P. (2015) *NIST Releases SHA-3 Cryptographic Hash Standard, NIST*. Available at: https://www.nist.gov/news-events/news/2015/08/nist-releases-sha-3-cryptographic-hash-standard (Accessed: 24 May 2021).

Heron, S. (2009) ‘Advanced Encryption Standard (AES)’, *Network Security*, 2009(12), pp. 8–12. doi: 10.1016/S1353-4858(10)70006-4.

Johnson, D. and Menezes, A. (1999) *The Elliptic Curve Digital Signature Algorithm (ECDSA)*.

Kwon, A., Lu, D. and Devadas, S. (2020) ‘XRD: Scalable Messaging System with Cryptographic Privacy’, p. 18.

Lepsoy, J. *et al.* (2015) ‘Metadata protection scheme for JPEG privacy security using hierarchical and group-based models’, in *2015 5th International Conference on Information Communication Technology and Accessibility (ICTA)*, pp. 1–5. doi: 10.1109/ICTA.2015.7426905.

Mahajan, D. P. and Sachdeva, A. (2013) ‘A Study of Encryption Algorithms AES, DES and RSA for Security’, *Global Journal of Computer Science and Technology*. Available at: https://computerresearch.org/index.php/computer/article/view/272 (Accessed: 15 December 2020).

Mahto, D. and Yadav, D. K. (2017) ‘RSA and ECC: A Comparative Analysis’, 12(19), p. 9.

Maletsky, K. (2020) ‘RSA vs. ECC Comparison for Embedded Systems’, p. 6.

Mamta *et al.* (2020) ‘Metadata Security Measures for Protecting Confidential Information on the Cloud’, in Chellappan, S., Choo, K.-K. R., and Phan, N. (eds) *Computational Data and Social Networks*. Cham: Springer International Publishing, pp. 398–410.

Rivest (1990) *The MD4 Message Digest Algorithm*. Available at: https://datatracker.ietf.org/doc/html/rfc1186 (Accessed: 15 June 2021).

Schneier, B. (1996) *Applied cryptography : protocols, algorithms, and source code in C*. New York : Wiley. Available at: http://archive.org/details/Applied_Cryptography_2nd_ed._B._Schneier (Accessed: 10 June 2021).

Siahaan, A. P. U. and Oktaviana, B. (2018) ‘Comparative Analysis of RSA and ElGamal Cryptographic Public-key Algorithms’, p. 10.

Simmons, G. J. (1979) ‘Symmetric and Asymmetric Encryption’, *ACM Computing Surveys*, 11(4), pp. 305–330. doi: 10.1145/356789.356793.

Swenson (2012) *NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition*, *NIST*. doi: 10/nist-selects-winner-secure-hash-algorithm-sha-3-competition.

Wei Dai (2009) *Speed Comparison of Popular Crypto Algorithms*. Available at: https://www.cryptopp.com/benchmarks.html (Accessed: 24 May 2021).

Wheeler, D. J. and Needham, R. M. (1995) ‘TEA, a tiny encryption algorithm’, in Preneel, B. (ed.) *Fast Software Encryption*. Berlin, Heidelberg: Springer Berlin Heidelberg (Lecture Notes in Computer Science), pp. 363–366. doi: 10.1007/3-540-60590-8_29.

Wheeler J. David and Rojer M. Needham (1998) ‘Correction to XTEA’. Available at: http://www.movable-type.co.uk/scripts/xxtea.pdf (Accessed: 22 May 2021).

Wijayanto, H., Riadi, I. and Prayudi, Y. (2016) ‘Encryption EXIF Metadata for Protection Photographic Image of Copyright Piracy’, 5(5), p. 7.

Yarrkov, E. (2010) *Cryptanalysis of XXTEA*. 254. Available at: https://eprint.iacr.org/2010/254 (Accessed: 22 May 2021).